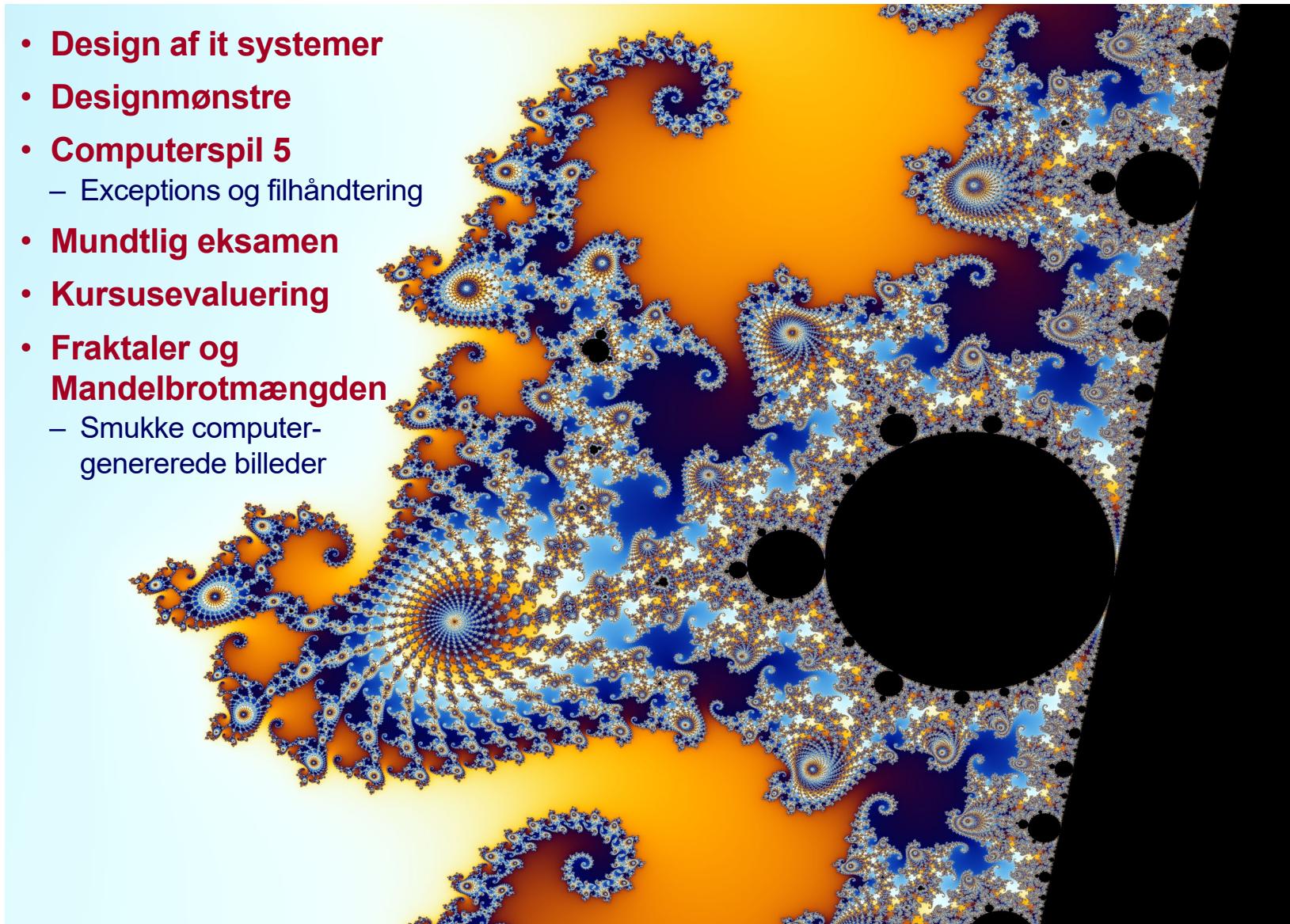


Forelæsning Uge 15

- Design af it systemer
- Designmønstre
- Computerspil 5
 - Exceptions og filhåndtering
- Mundtlig eksamen
- Kursusevaluering
- Fraktaler og
Mandelbrotmængden
 - Smukke computer-genererede billeder



● Design af it systemer

- I dette kursus har vi primært beskæftiget os med, hvordan man laver gode klasser
 - Skal være lette at bruge, forstå og vedligeholde
 - I alle jeres opgaver har I på forhånd vidst, hvilke klasser der skulle være i jeres projekt
 - Men sådan er det selvfølgelig ikke i virkeligheden
- De faser, hvor systemets overordnede opbygning fastlægges kaldes analyse og design
 - I analysefasen undersøger vi det kommende system
 - I designfasen fastlægger vi dets **arkitektur** – dvs. hvilke klasser, der skal være, og hvordan de interagerer med hinanden
 - Stort og komplekst problemområde
 - Vi vil kun se på nogle af de helt grundlæggende principper og teknikker
- I store systemudviklingsprojekter bruger man mere tid på analyse og design end man bruger på implementation
 - Tid brugt på omhyggelig analyse og godt design, spares (mange gange), når man senere skal programmere, afteste og vedligeholde systemet

Navneord og udsagnsord

- For små systemer kan det være en hjælp at gennemse tekstuelle beskrivelser af systemet
 - Navneord vil så ofte svare til **klasser** og **objekter** (eller **feltvariabler**)
 - Tilsvarende vil udsagnsord svare til handlinger, dvs. **metoder**
- Eksempel: Reservationssystem til biograf

Navneord

The cinema booking system should store **seat bookings** for multiple **theaters**. Each theater has **seats** arranged in **rows**. **Customers** can reserve seats and are given a **row number** and **seat number**. They may request bookings of several adjoining seats. Each booking is for a particular **show** (i.e., the screening of a given **movie** at a certain time). Shows are at an assigned **date** and **time**, and scheduled in a theater where they are screened. The system stores the customer's **phone number**.

Udsagnsord

The cinema booking system should **store seat bookings** for multiple theaters. Each theater **has seats** arranged in rows. Customers can **reserve seats** and **are given a row number and seat number**. They may **request bookings** of several adjoining seats. Each booking is for a particular show (i.e., the screening of a given movie at a certain time). Shows are at an assigned date and time, and **scheduled in a theater** where they are screened. The system **stores the customer's phone number**.

Klasser og metoder

Klasser og metoder

Cinema booking system

stores (seat bookings)

stores (phone numbers)

Theater

has (seats)

Customer

reserves (seats)

is given (row number, seat number)

requests (seat booking)

Show

is scheduled (in theater)

Simple typer

Movie

Time

Date

Seat number

Row number

Phone number

- **Teknikken er forbavsende effektiv**
 - Hjælper os til at komme i gang
 - Vi kan så lave justeringer og tilpasninger undervejs

Ansvarsområder og samarbejdspartnere

- Næste skridt er at identificere de enkelte klassers ansvarsområder og samarbejdspartnere
 - Til dette formål bruges såkaldte CRC-kort, hvor vi har et kort for hver klasse ($\text{CRC} \approx \text{Class} + \text{Responsibilities} + \text{Collaborators}$) **Klassens samarbejdspartnere**
(de klasser den interagerer med)

Klassens navn →	CinemaBookingSystem	Collaborators
Klassens ansvarsområder (bliver til metoder) →	Can find shows by title and day Stores collection of shows Retrieves/displays show details ...	Show Collection

- **CRC-kortene udfyldes ved at gennemspille en række scenarier – dvs. eksempler på handlingssekvenser, der kan forekomme i systemet**
 - Reservation af plads til "Godfarther i aften"
 - 5 pladser ved siden af hinanden
 - Oplysning om, hvilke pladser en kunde har reserveret
 - Tilføjelse af 2 ekstra pladser til eksisterende reservation
 - Afbestilling, osv.

Se evt. Taxi Company
eksemplet i kapitel 16

● Designmønstre

- **Tidligere i kurset er vi stødt på algoritmeskabeloner**
 - De hjalp os til hurtigt at kunne programmere metoder for diverse søgninger, ved at **genbruge** eksisterende **mønstre** fastlagt i skabelonerne
- **Ideen i designmønstre (design patterns) er den samme**
 - Men nu arbejder vi ikke på enkelte metoder
 - Et designmønster beskriver, hvordan et **antal klasser** arbejder sammen om at løse et komplekst problem
- **Beskrivelsen af et designmønster består af**
 - Mønsterets **navn**
 - Beskrivelse af **problemet**
 - Beskrivelse af **løsning** – ofte i form af et UML klassediagram
 - **Fordele** og **ulemper** ved at anvende mønsteret

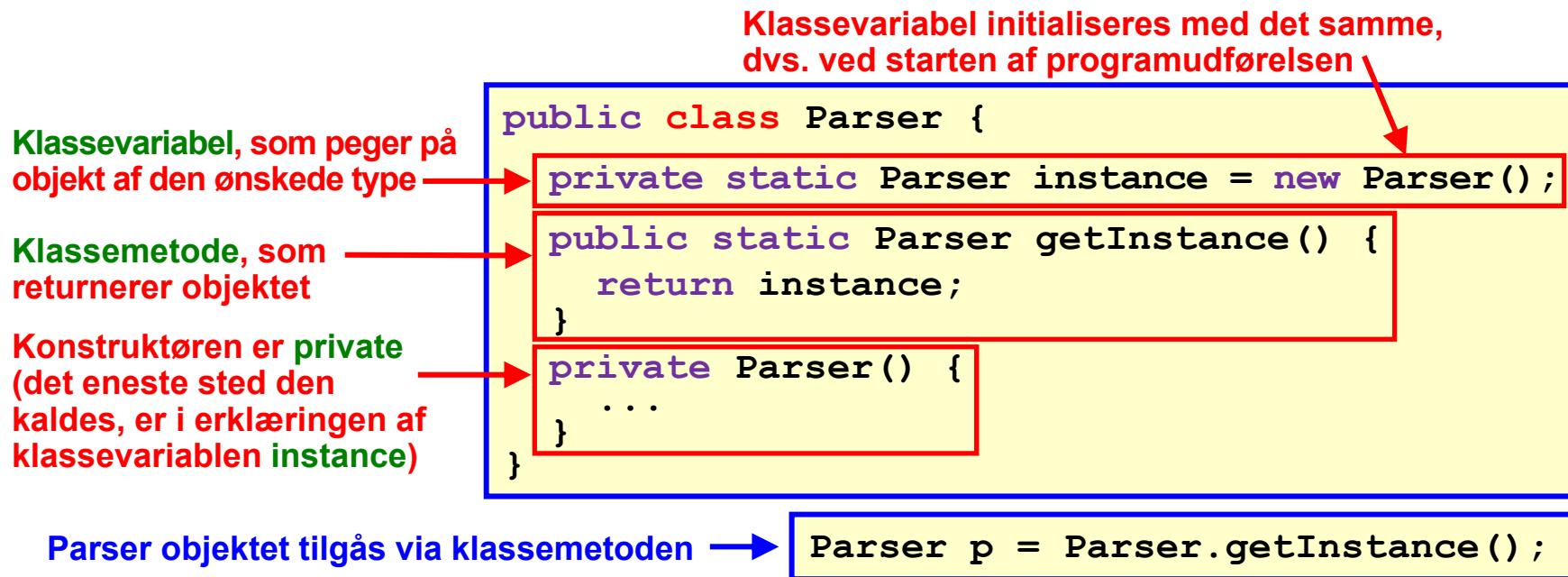
Anvendelse af designmønstre

- Designmønstre anviser gode løsninger på problemer, der er så generelle, at vi støder på dem igen og igen
 - Vi genbruger ikke specifik Java kode, men mønsteret, dvs. idéen om, hvordan klasserne arbejder sammen om at løse det givne problem
- Designmønstre giver os desuden et fagsprog, hvormed vi kan beskrive vores design
 - Hvis en udvikler, f.eks. siger: "Her bør vi bruge en Singleton" – vil de andre systemudviklere straks vide, hvad hun mener
 - Singleton er nemlig navnet på et meget kendt og udbredt designmønster (som vi skal stifte nærmere bekendtskab med på næste slide)
- Designmønstre blev for alvor kendt via bogen "Design Patterns: Elements of Reusable Object-Oriented Software" fra 1995
 - Den beskriver ca. 50 forskellige designmønstre
 - En god software designer vil kende (og bruge) mange af disse
 - Prøv at Google "design pattern"

Singleton mønsteret

- **Problemet**
 - Vi har en klasse, hvor vi ønsker, at der kun skal kunne instantieres et enkelt objekt – som mange forskellige klasser/objekter har adgang til
- **Løsningen**
 - Erklær en klassevariabel, som peger på et objekt af den ønskede type
 - Erklær en klassemethode, som returnerer ovenstående objekt
 - Klassens konstruktør er private, og dermed kan der ikke skabes andre objekter end det, som klassevariablen peger på
- **Eksempler**
 - World-of-zuul projektet har én parser til inputtekst
 - I Skildpadde opgaverne var Canvas klassen implementeret som en Singleton (vi kan skabe flere skildpadder, men alle tegner på samme lærred)

Singleton mønsteret (implementation)



- **Bemærk**

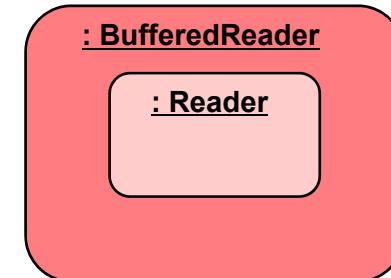
- Hvis `getInstance` havde været en almindelig instansmetode skulle man have Parser objektet for at kunne kalde `getInstance` (hvilket ikke giver mening)
`Parser p = p.getInstance();`
- Hvis `instance` havde været en feltvariabel ville `getInstance` ikke have adgang til den og derudover ville der aldrig blive lavet et Parser objekt

Alternativt kan man undlade klassemetoden og i stedet gøre klassevariablen public og final

Man kan også implementere en Singleton som en enumeration, der kun har ét element

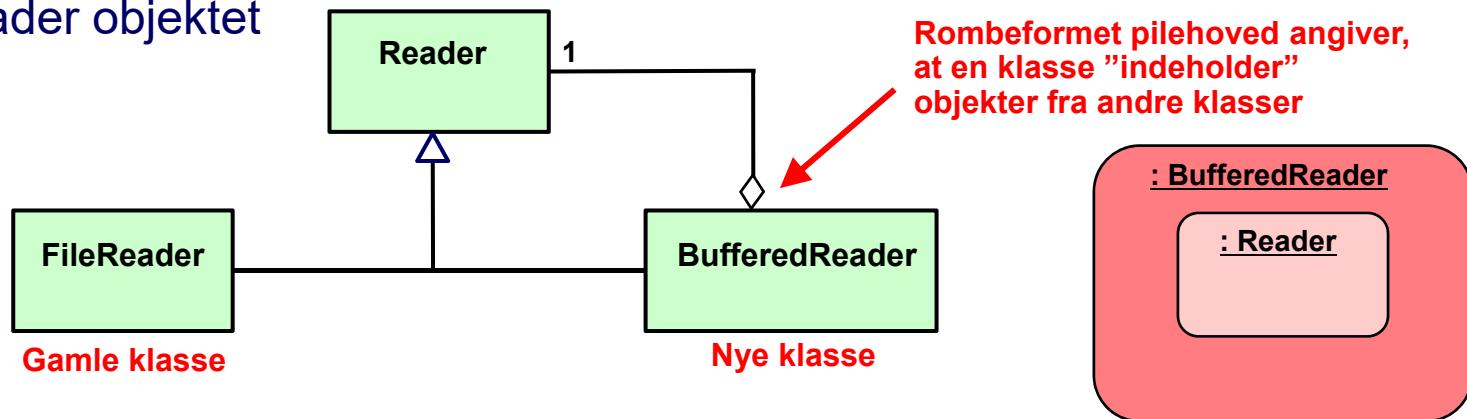
Decorator mønsteret

- **Problemet**
 - Vi ønsker at udvide funktionaliteten af en eksisterende klasse – uden at ændre dens interface (eksisterende metoder)
- **Løsningen**
 - Der laves en ny klasse, der set fra brugerens synspunkt, erstatter den gamle
 - Et objekt af den nye klasse "indeholder" et objekt af den gamle klasse
 - Kald af en metode i et objekt af den nye klasse sendes videre til objektet af den gamle klasse. Herudover udføres, der eventuelt noget ekstra (dekorationen)
- **Eksempel**
 - BufferedReader har de samme metoder som Reader, men tilføjer en ekstra metode til at læse en hel linje ad gangen
 - Ethvert BufferedReader objekt "indeholder" et Reader objekt, som alle "gamle" metodekald sendes videre til



Decorator mønsteret (BufferedReader)

- **UML diagram**
 - BufferedReader objektet ”indeholder” et Reader objekt (som i det viste UML diagram er et FileReader objekt, men kan være et vilkårligt Reader objekt)
 - BufferedReader er en subklasse af Reader, og BufferedReader objektet kan derfor (ligesom FileReader objektet) anvendes, hvor der skal bruges et Reader objektet



- **Instantiering**
 - Det objekt, der skal dekoreres, gives med som parameter til konstruktøren

```
BufferedReader in = new BufferedReader(new FileReader("System.in"));
```

Nye klasse **Gamle klasse**

Decorator mønsteret (BufferedReader fortsat)

- Skitse til implementation af BufferedReader

```
public class BufferedReader extends Reader {  
    private Reader reader;  
  
    public BufferedReader(Reader reader) {  
        super();  
        this.reader = reader;  
        ...  
    }  
  
    // Reads a single character  
    public void read() throws IOException {  
        reader.read();  
    }  
    ...  
    // Closes the stream  
    public void close() throws IOException {  
        reader.close();  
    }  
  
    // Reads a line of text  
    public void readLine() throws IOException {  
        ...  
    }  
}
```

Feltvariabel

Konstruktøren gemmer det "gamle" objekt i feltvariablen reader

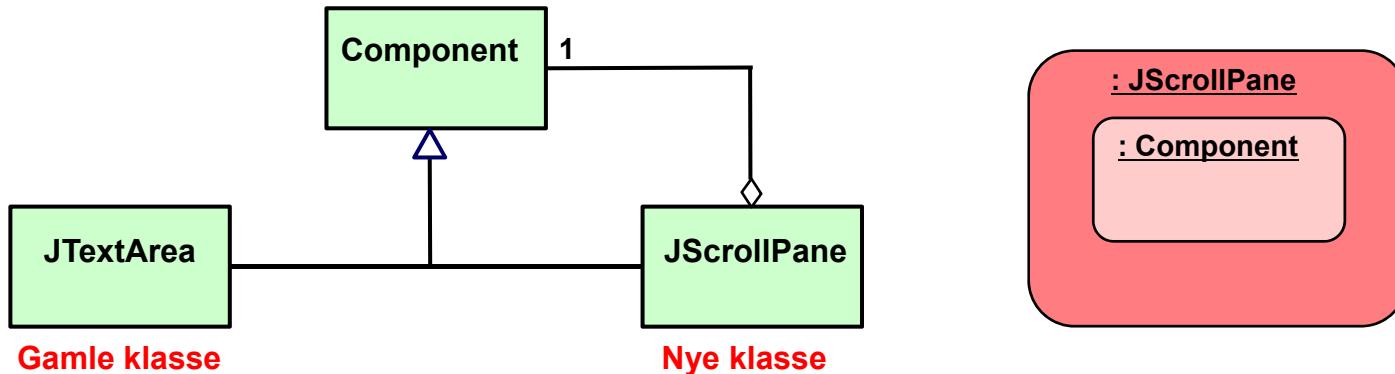
Kald af "gamle" metoder sendes videre til det objekt, som feltvariablen reader peger på

Ny metode (implementeres fra scratch)

Den "nye" klasse er en subklasse af Reader (ligesom den "gamle")
Den kan derfor bruges alle de steder, hvor man skal bruge et Reader objekt

Decorator mønsteret (andet eksempel)

- **Klassen JScrollPane kan dekorere (forsyne) et vilkårligt Component objekt (i en GUI) med scrollbarer**
 - JScrollPane objektet "indeholder" det oprindelige Component objekt
 - JScrollPane er en subklasse af Component klassen og det nye objekt kan derfor (ligesom det gamle) anvendes, hvor der skal bruges et Component objekt
 - Kald af "gamle" metoder sendes videre til objektet af den gamle klasse, men herudover udføres der ting, der håndterer de tilføjede scrollbarer (dekorationen)



- For et JTextArea objekt ser instantieringen ud som følger

```
JScrollPane scrollArea = new JScrollPane(new JTextArea(10, 25));
```

I Computerspil 5 opgave 4, skal I lave et tekstfelt med scrollbarer

Nye klasse
↑

Objekt fra den
gamle klasse
↑

Decorator mønsteret kontra nedarvning

- I stedet for at bruge Decorator mønsteret kunne man have udvidet klassen ved hjælp af nedarvning
 - F.eks. kunne vi have lavet en BufferedReader klasse, der er en subklasse af FileReader
- Brug af Decorator mønsteret er imidlertid langt mere generelt
 - Reader klassen har mange forskellige subklasser, og ved hjælp af Decorator mønsteret har vi "i et hug" dekoreret dem alle (ved hjælp af BufferedReader)
 - Analogt har vi "i et hug" dekoreret alle Component subklasser (ved hjælp af JScrollPane)
- Herudover er Decorator mønsteret dynamisk
 - Det tillader, at man på et vilkårligt tidspunkt under programudførelsen kan dekorere et eksisterende objekt og f.eks. forsyne det med scrollbarer
 - Det kan man ikke, hvis man bruger nedarvning, idet et eksisterende objekt ikke kan skifte type til en subklasse

Factory method mønsteret

- **Problemet**
 - Vi har en række klasser, og vil gerne have en **ensartet** måde at skabe nye objekter fra disse klasser
- **Løsningen**
 - Vi definerer et **interface** med en **factory metode**, der kan skabe de nye objekter
 - Typen på de skabte objekter, afhænger af det objekt, der kalder factory metoden
- **Eksempel**
 - I kapitel 12 skabte vi et antal ræve og et antal kaniner
 - Vi kan definere et **ActorFactory interface** og tilhørende klasser **FoxFactory** og **RabbitFactory**, som implementerer interfacet (en klasse for hver Actor)
 - Simulator objektet gemmer en liste af ActorFactory objekter, og kan bede hver af disse om at skabe et antal objekter (nye dyr af den pågældende slags)

Factory method mønsteret (Iterator)

- Alle Collection typer har en iterator metode, der returnerer en iterator for den pågældende Collection type
 - Collection typen implementerer Iterable interfacet ved at implementere en iterator metode (som er vores factory method)
 - iterator metoden returnerer en iterator, som kan bruges til at gennemløbe den pågældende objektsamling

```
Iterator<E> it = coll.iterator();
```

coll kan f.eks være af typen
ArrayList<String> eller
HashSet<Person>
(men ikke HashMap<K,V>)

- Vi kan lave generiske metoder, som kan behandle alle slags objektsamlinger

```
public <E> void process (Collection<E> coll) {  
    Iterator<E> it = coll.iterator();  
    ...  
}
```

Type

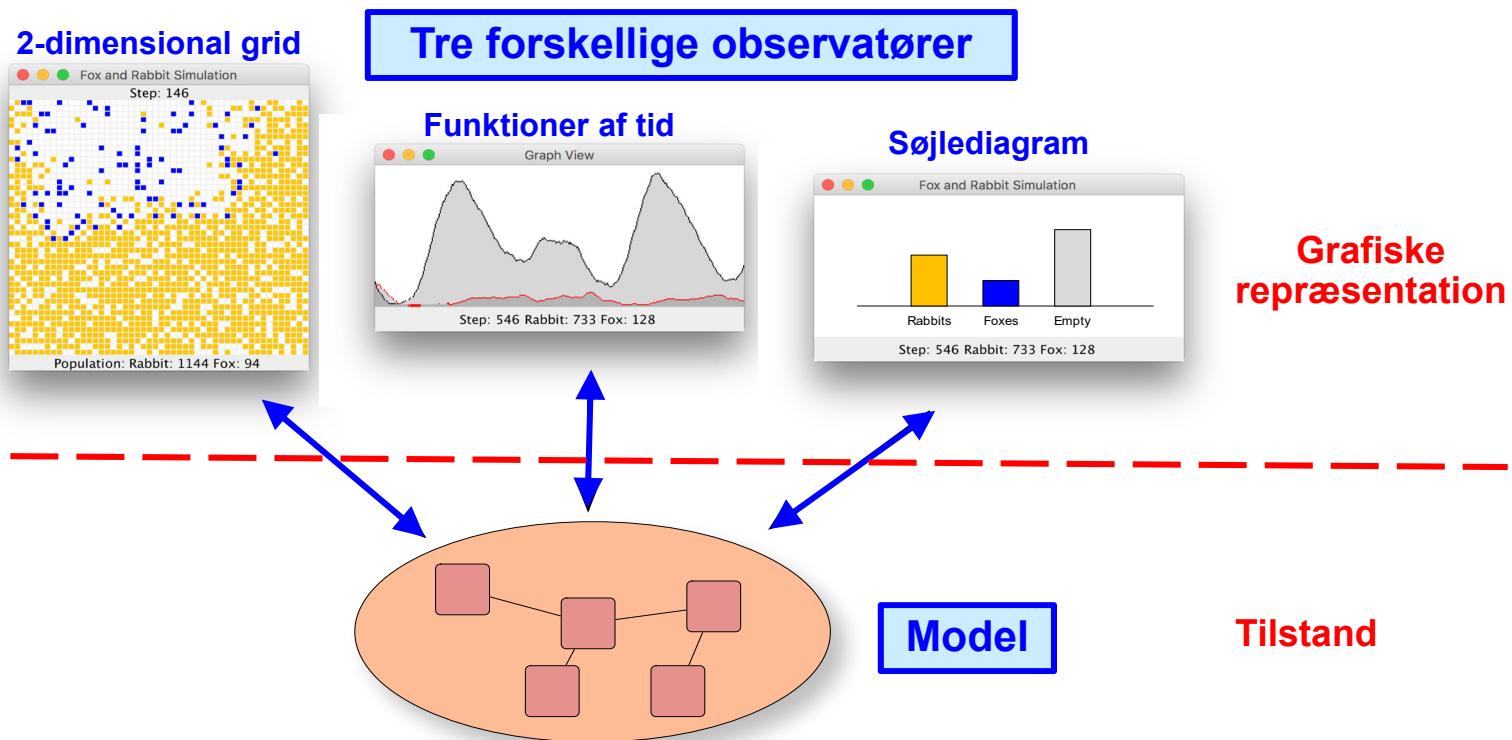
Parameteren kan være en
vilkårlig objektsamling

Ved hjælp af factory metoden iterator, får vi en iterator,
som kan bruges til at gennemløbe objektsamlingen

Observer mønsteret

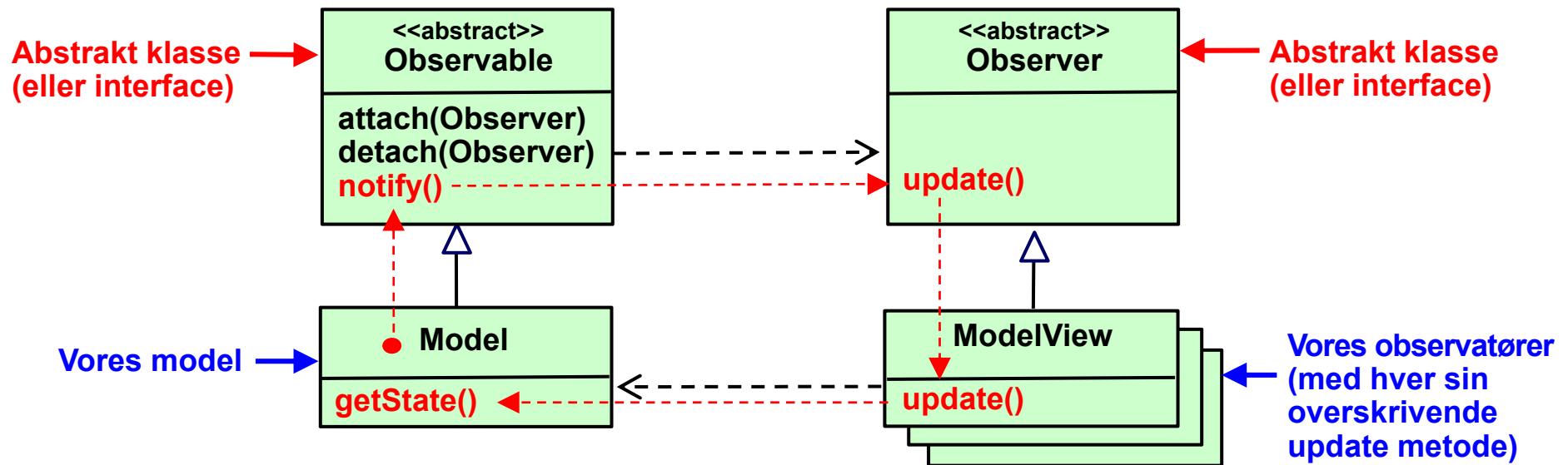
- **Problemet**

- Vi ønsker at **adskille** et systems interne operationer og tilstand fra dets repræsentation på skærmen (model/view separation)
- Dette vil gøre det lettere, at tilknytte flere forskellige repræsentationer
- Observatørerne **gemmer intet** om systemets tilstand. Denne information får de fra modellen



Observer mønsteret (fortsat)

- Løsningen



- **Observable** har en **attach** metode, som tillader de forskellige **Observer** objekter at tilmelde sig (analogt til `addActionListener`)
- Når modellens tilstand ændres, kalder **Model** objektet den nedarvede **notify** metode fra `Observable` klassen
- **notify** metoden kalder **update** metoderne i de **tilmeldte** `Observer` objekter (via dynamic method lookup)
- **update** metoderne (i de tilmeldte `Observer` objekter) bruger accessormetoden **getState** til at få information om modellens opdaterede tilstand

● Afleveringsopgave: Computerspil 5

- **I den femte og sidste delaflevering skal I bruge nogle af de ting, som I har lært om exceptions og fil-baseret input/output**
 - I skal lave en **Log** klasse, der kan gemme et spil, dvs.
 - de valg som brugeren laver under et spil
 - seed-værdien for det Random objekt, der styrer de automatiske spillere
 - de optionsværdier der gælder under spillet
 - Dernæst skal I skrive kode, som kan gemme et Log objekt i fil systemet og senere genindlæses filen, hvorpå det optagne spil kan afspilles
 - De fejl, der kan opstå, når I gemmer og genindlæser en fil, skal håndteres ved hjælp af exceptions
- **Herudover skal I tilføje et tekstfelt, hvori I lister de træk, som brugeren laver under spillet**
 - Tekstfeltet skal forsynes med scrollbarer, som beskrevet under gennemgangen af **Decorator** designmønsteret
- **Vigtigt at afleveringsfristen den 11. december kl. 13.00 overholdes**
 - Afleveringer efter mandag kl. 18 rettes ikke og får dermed automatisk 0 point
 - Der vil ikke være mulighed for genafleveringer

Pause

● Mundtlig eksamen

- **Ved den mundtlige eksamen forventer vi, at du demonstrerer**
 - Kendskab til de **vigtigste begreber** inden for det trukne emneområde
 - Evne til at **programmere i Java** ved at præsentere og forklare små velvalgte programstumper indenfor emneområdet
 - Evne til at **svare på spørgsmål** inden for emneområdet, herunder relatere kursets projekttopgaver til emneområdet
- **Varighed**
 - Du er ”på scenen” i ca. 15 minutter
 - De næste 3-5 minutter bruges til voting, meddelelse og forklaring af din karakter samt skift til næste eksaminand
- **En af de mest effektive måder at træne til eksamen, er at høre andre studerende (og lære af deres gode og dårlige ting)**
 - Vi opfordrer derfor kraftigt til, at I går ind og hører nogle af jeres medstuderende
 - Nogle synes, at det er ”upassende” – men faktisk vil det for langt de fleste eksaminander være betryggende, at der er ”neutrale” tilhørere tilstede under eksaminationen

Forløbet af eksamen

- **Præsentationen**

- De første 2-3 minutter får du lov til at skrive din disposition på tavlen og snakke uforstyrret (indtil den værste nervøsitet har lagt sig)
- Derefter vil eksaminator/censor afbryde med forskellige spørgsmål for at hjælpe dig med
 - at rette eventuelle småfejl
 - at få dækket de vigtigste ting indenfor emneområdet
- Ved at stille spørgsmål tjekker vi også, om du har forstået stoffet eller blot lært det udenad
- Jo bedre du har forberedt dig og jo mere initiativ du selv udviser – jo bedre har du styr på, hvor du ”kommer hen” under eksamen (f.eks. hvilke programmeringseksempler, du skal gennemgå)

- **Der gives karakter efter 12-skalaen**

- Pointene fra køreprøven og computerspilsopgaven tæller med i fastlæggelsen af den endelig karakter for kurset (tæller tilsammen med 25%)
- I praksis betyder det, at høje point kan trække en karakter op, mens lave point kan trække en karakter ned
- Uanset pointtal kan du dumpe, hvis den mundtlige præstation er uacceptabel

Forberedelse til mundtlig eksamen

- **Disposition**

- For hvert af de 9 spørgsmål laves en kort velgennemtænkt disposition
- A4-ark med 10-20 ord (ingen figurer, formler, programstumper, eller lignende)
 - Opremser de begreber og eksempler, som du vil bruge
- Til eksamen starter du med at skrive dispositionen op i et hjørne af tavlen
 - Dulmer ofte den værste nervositet
 - Herefter lægges dispositionen **helt væk** (eller med bagsiden opad)
- Du får ikke point for at kunne læse op af dispositionen
 - Derfor skal den være så kort som overhovedet mulig
 - Den er blot en huskeliste over, hvad du vil præsentere og i hvilken rækkefølge
 - Du bør kunne præsentere de ting, som du har udvalgt i dispositionen på ca. 10 min
 - Resten af tiden bruges til at svare på diverse spørgsmål inden for det trukne emne samt resten af pensum

Java kode eksempler

- **Valg af eksempler**
 - Det er vigtigt, at du **på forhånd**, for hvert af de 9 spørgsmål, har valgt nogle gode eksempler på Java kode, som du vil præsentere
 - Eksemplerne kan være ”stjålet” fra lærebogen, mine slides eller nogle af de opgaver, som du har lavet på kurset
 - Brug tid på at finde gode eksempler, og tid på at træne i at præsentere dem
 - Eksemplerne skal være korte og tydeligt vise de ting, som er centrale for din præsentation
- **Hvis du ikke selv forbereder små velvalgte Java eksempler, finder vi nogle, som du skal præsentere**
 - Det gør ikke opgaven lettere
- **Det er tilladt at lave sine egne eksempler eller finde dem på nettet**
 - Dette anbefales dog **ikke**.
 - Dels skal du bruge længere tid på at forklare dem (da eksinator og censor ikke kender dem i forvejen)
 - Dels kan du ikke så godt få hjælp (fra eksinator og censor), hvis der opstår problemer under præsentationen

Træning gør mester

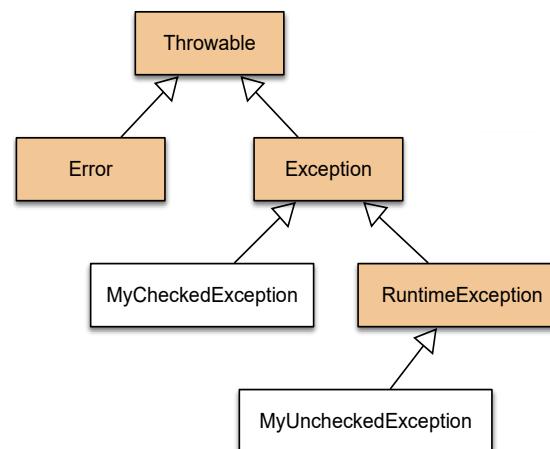
- **Hvert spørgsmål bør trænes mindst 5 gange**
 - Træningen skal være så realistisk som overhovedet muligt
 - Det er ikke nok at tænke på, hvad du vil sige og skrive
 - Du skal formulere sætningerne og sige dem højt
 - Du skal skrive tingene på et whiteboard eller et stykke papir
- **Eksamens er ikke en test i skønskrift – men det er oplagt en fordel at eksaminator og censor kan læse det, som du skriver**
 - Det er forbavsende svært at skrive læseligt på et whiteboard (øv dig i det)
 - Hold fornuftig tavleorden
 - Du må gerne forkorte lange navne og lignende og bruge gentagelsestegn/ streger (øv dig i at gøre det på en god måde)
 - Visk ikke noget ud (bortset fra smårettelser)
 - Eksaminator og censor tæller alt det med, som du har skrevet og sagt
- **Hold dig til dispositionen**
 - Lad være med at improvisere undervejs
 - Opfind ikke nye eksempler, som du ikke har gennemtænkt
- **Under de sidste træninger bør du ved hjælp af et ur tjekke, at du har stof nok til 10 minutter – hverken mere eller mindre**
 - De resterende 5 minutter af eksaminationen bruges til at besvare spørgsmål fra eksaminator og censor

Gode råd omkring eksamen

- **Husk at præsentere de overordnede begreber – inden du kaster dig ud i detaljeret Java kode**
 - Fx vil det for spørgsmålet om grafiske brugergrænseflader være rimeligt at starte med at snakke om AWT, Swing og JavaFX, og at man vil bruge en kombination af de to første



- Analogt vil det for exceptions være en god ide at opridse figuren, der viser forskellige exception typer, og bruge figuren til at beskrive, hvordan oversætteren skelner mellem checked og unchecked exceptions.



Gode råd omkring eksamen (fortsat)

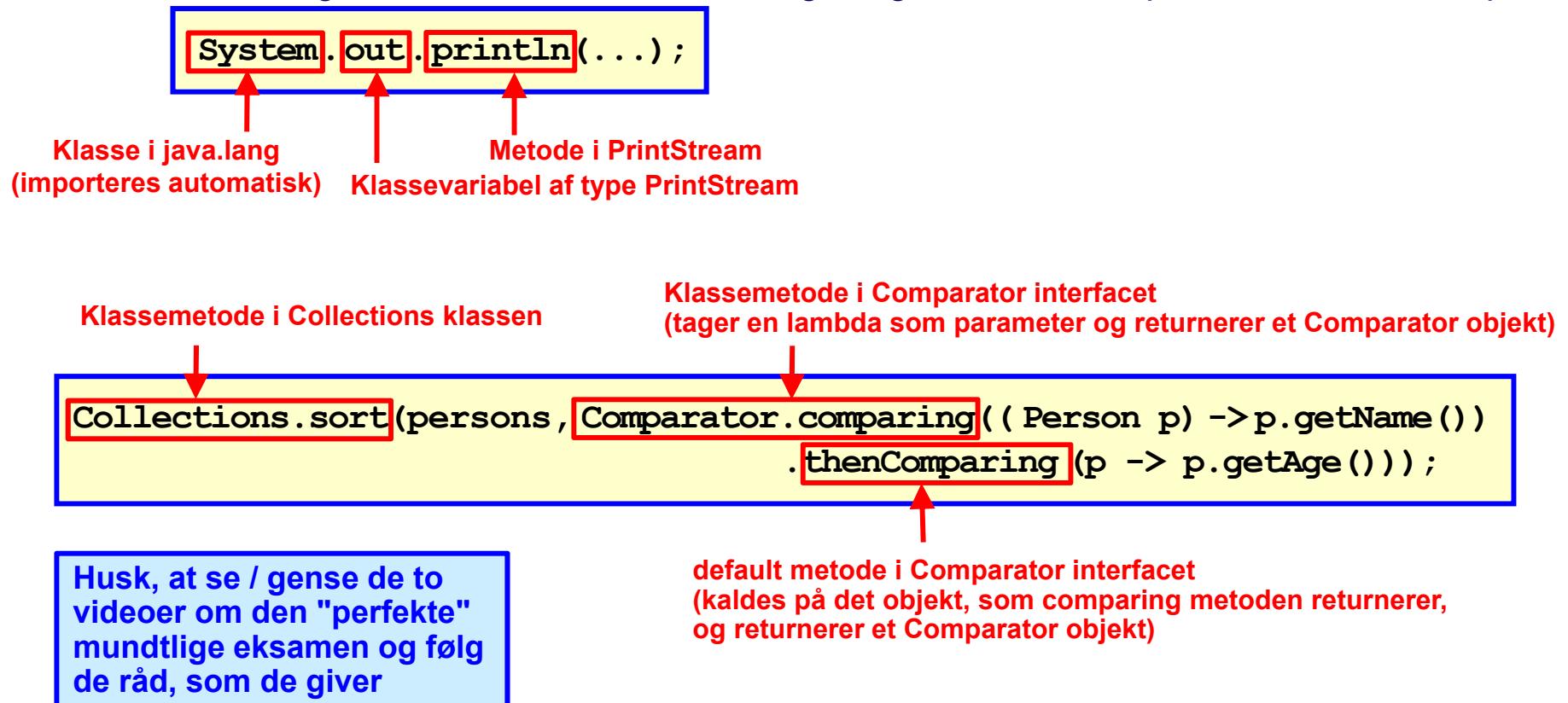
- **Vis at du forstår, hvad der ligger bag de navne vi bruger i bogen / på kurset**
 - Sig f.eks. at det er oversætteren, der tjekker en **checked** exception, og at det den tjekker er, at man på kaldstedet håndterer den rejste exception (enten ved at placere metodekaldet i en try-catch statement eller ved at videresende den rejste exception til omgivelserne ved at tilføje throws i metodens header)
 - Analogt bør man forklare, at **regression** tests er automatiske tests, der let kan gentages når man har ændret sin kode – for at undersøge om der skulle være sket **regression** (tilbageslag/forringelse) i form af, at man har introduceret nye fejl under ændringerne
- **Du må gerne inddrage relevante ting fra andre spørgsmål**
 - F.eks. vil det være helt relevant at nævne, at brug af subklasser er fortrinlig til at undgå **kodeduplikering**
 - Analogt bør man, når man snakker om ArrayList og HashMap, bruge **List** og **Map** som de statiske typer for sine variable, og fortælle at dette gør det lettere at skifte mellem forskellige implementationer af de to interfaces (ArrayList, LinkedList, HashMap, TreeMap, osv.)

Gode råd omkring eksamen (fortsat)

- **Brug ikke for megen tid på ting, der primært hører til andre spørgsmål**
 - F.eks. er det faktum, at man i en subklasses konstruktør, som det allerførste skal kalde superklassen konstruktør noget, der er generelt for nedarvning mellem klasser
 - Det hører derfor naturligt til spørgsmålet om nedarvning, mens det er mindre relevant, hvis det præsenteres under spørgsmålet om abstrakte klasser og interfaces
 - Her vil det være bedre at have tid til at snakke om funktionelle interface (hvad de er, og hvordan du har brugt dem i kurset) samt at snakke om, hvordan interfaces og abstrakte klasser bruges i Collection hierarkiet.

Du skal kunne forklare din kode

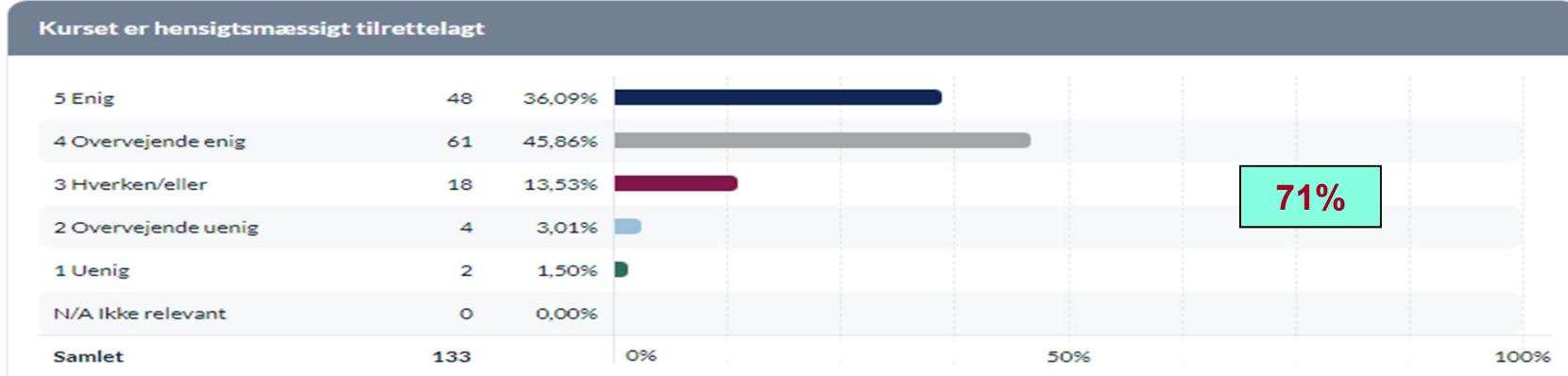
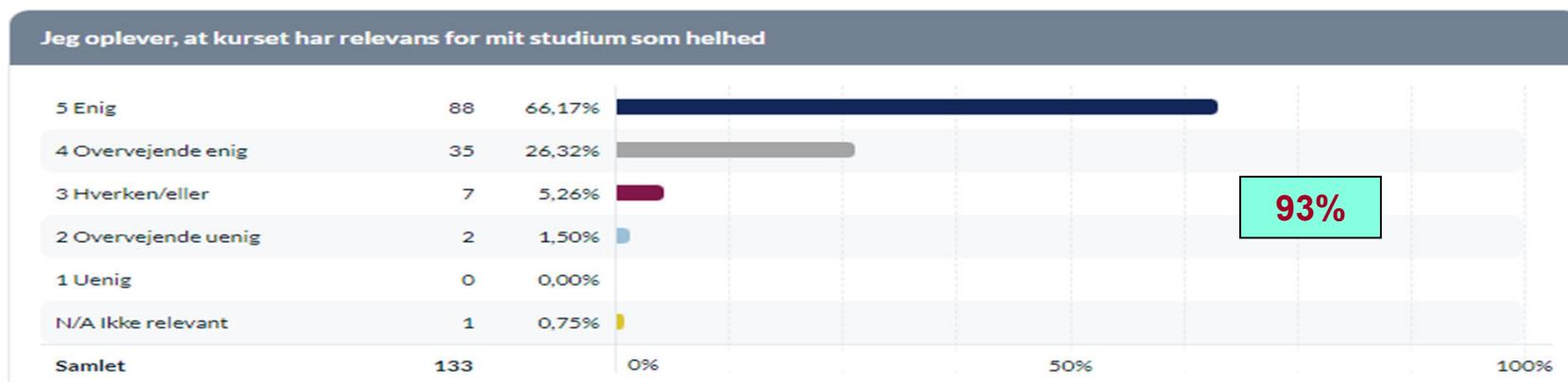
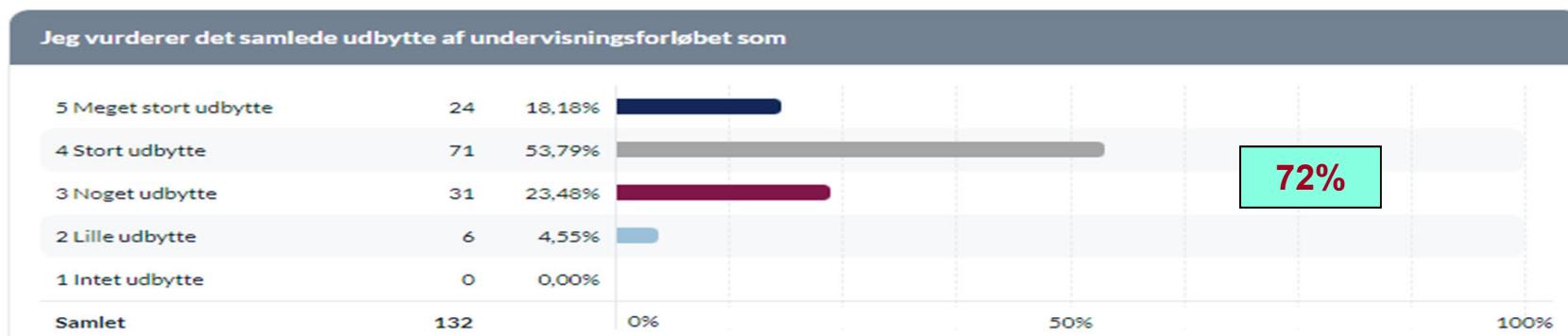
- Det er ikke nok at kunne skrive noget Java kode op
 - Du skal også vise, at du **forstår** koden
 - F. eks. skal du kunne forklare, hvorfor man bliver nødt til at bruge en klassemetode og en klassevariabel i Singleton mønsteret og have en privat konstruktør
 - Du skal også kunne forklare de forskellige ting i de kodestumper, som du skriver op



Forberedelse til eksamen

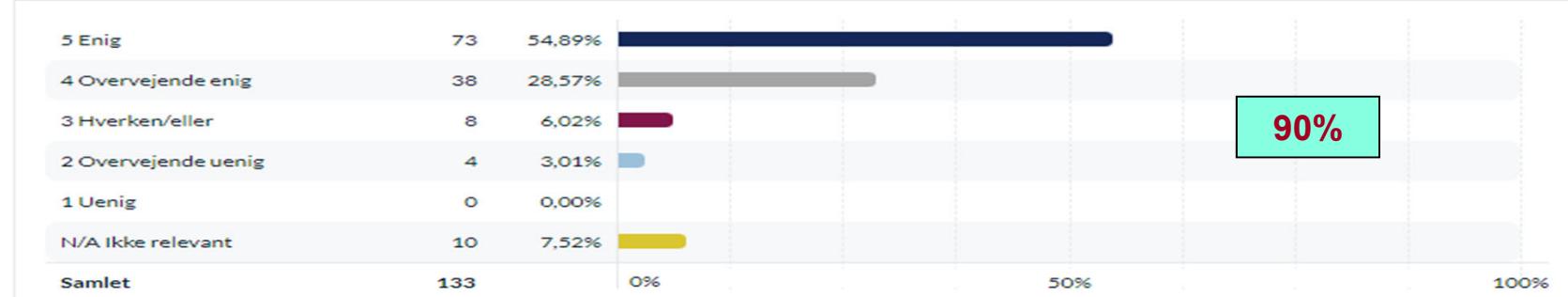
- **Spørgetimer i studiecaféen**
 - Sidste ordinære bemanding er mandag den 11. december kl 11-13 (den dag, hvor Computerspil 5 skal afleveres)
 - Mandag den 18. december, mandag den 8. januar og mandag den 15. januar vil der 11-13 være en instruktor fra kurset, der kan hjælpe med dispositioner og besvare spørgsmål om pensum mm
- **I kan selvfølgelig også fortsat bruge diskussionsforummet til at få råd og vejledning omkring dispositioner og andre ting**

● Kursusevaluering 2023 – 133 har svaret ≈ 75%

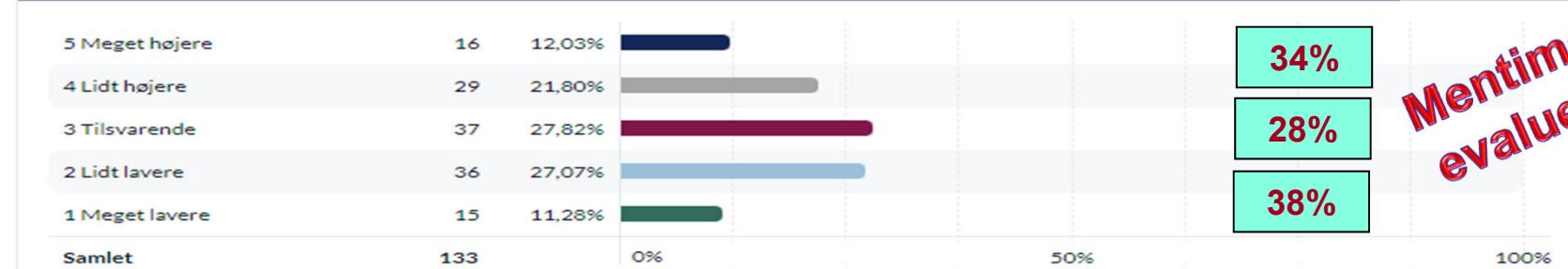


Kursusevaluering (fortsat)

Instruktoren/instruktørerne formidlede stoffet på en måde, der støttede min læring - angiv instruktors fulde navn i kommentarfeltet

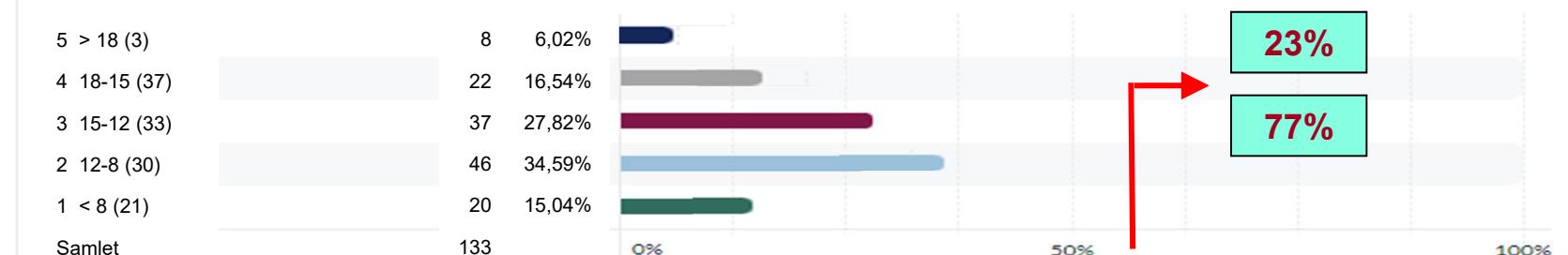


Jeg har prioriteret min arbejdsindsats på dette kursus ift. andre kurser på dette semester



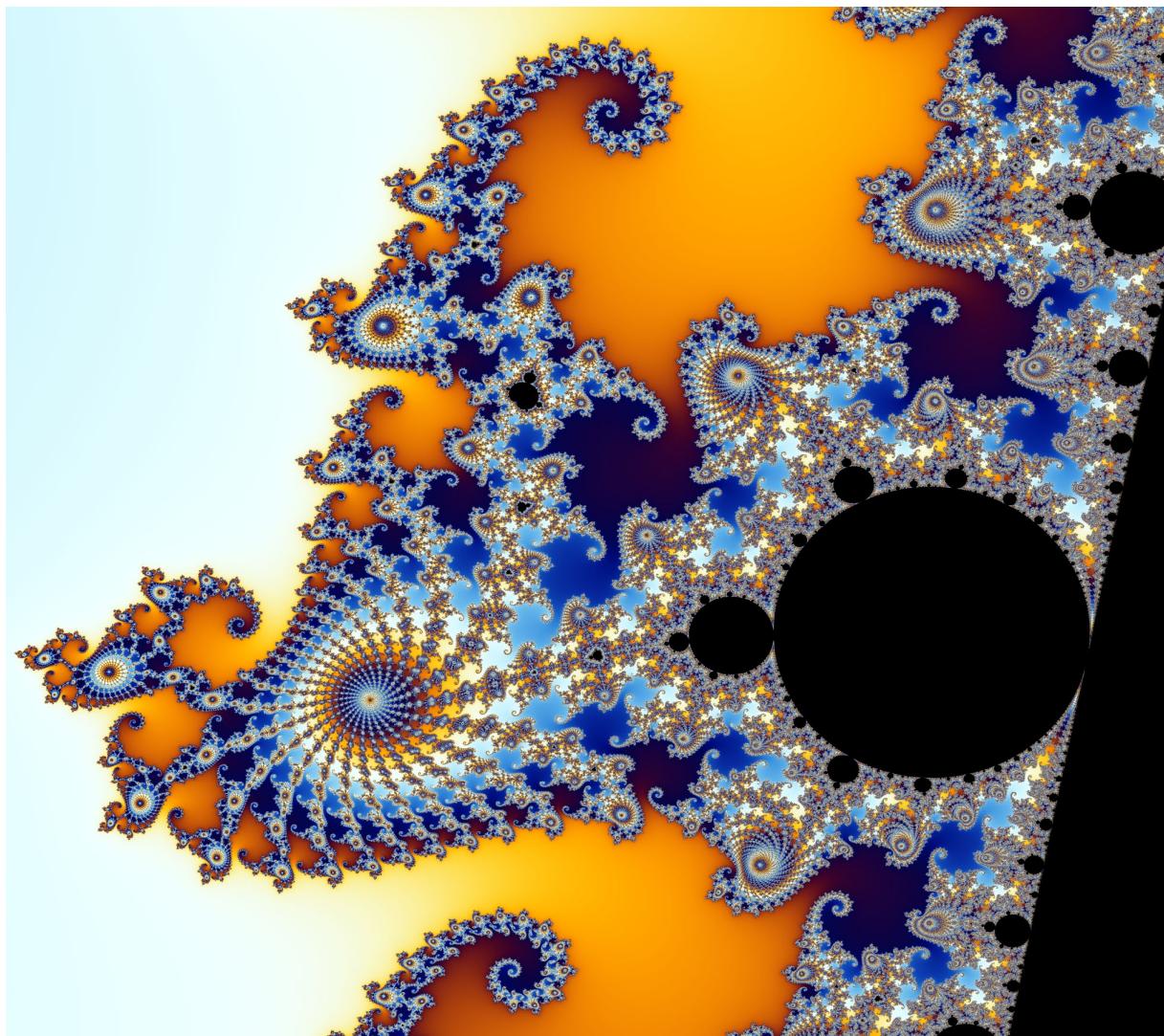
Mentimeter
evaluering

Samlet tidsforbrug pr uge



Normale tidsforbrug (15 timer)

● Fraktaler og Mandelbrotmængden



Benoit B. Mandelbrot

- 1924-2010
- Matematiker fra Yale University i USA

Ikke eksamenspensum

Kort gimmick, der viser,
hvad programmering
også kan bruges til

Definition af Mandelbrotmængden

- **Mandelbrotmængden er en mængde af punkter**
 - For et punkt **(a,b)** defineres en talfølge ved hjælp af nedenstående rekursive definition

$n = 0$

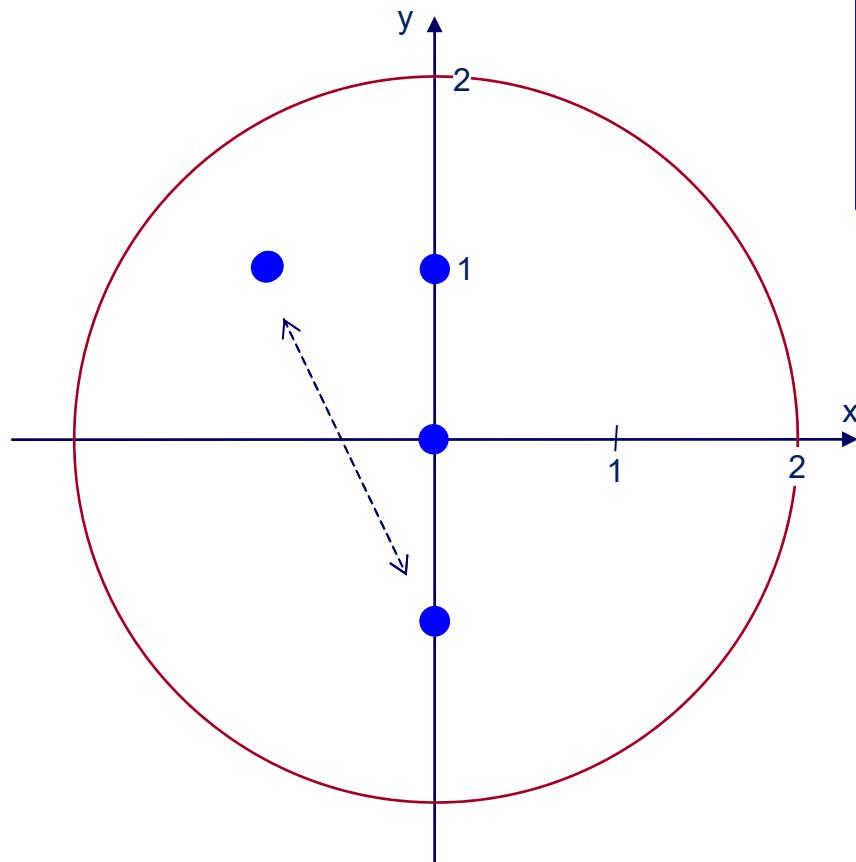
$$\begin{aligned}x_0 &= 0 \\y_0 &= 0\end{aligned}$$

$n \geq 0$

$$\begin{aligned}x_{n+1} &= x_n^2 - y_n^2 + a \\y_{n+1} &= 2 \times x_n \times y_n + b\end{aligned}$$

- Punktet **(a,b)** tilhører mandelbrotmængden, hvis og kun hvis punkternes afstand fra $(0,0)$ ikke går mod uendelig, når n vokser

Følgen for $(a,b) = (0, 1)$



$n = 0$

$$\begin{aligned}x_0 &= 0 \\y_0 &= 0\end{aligned}$$

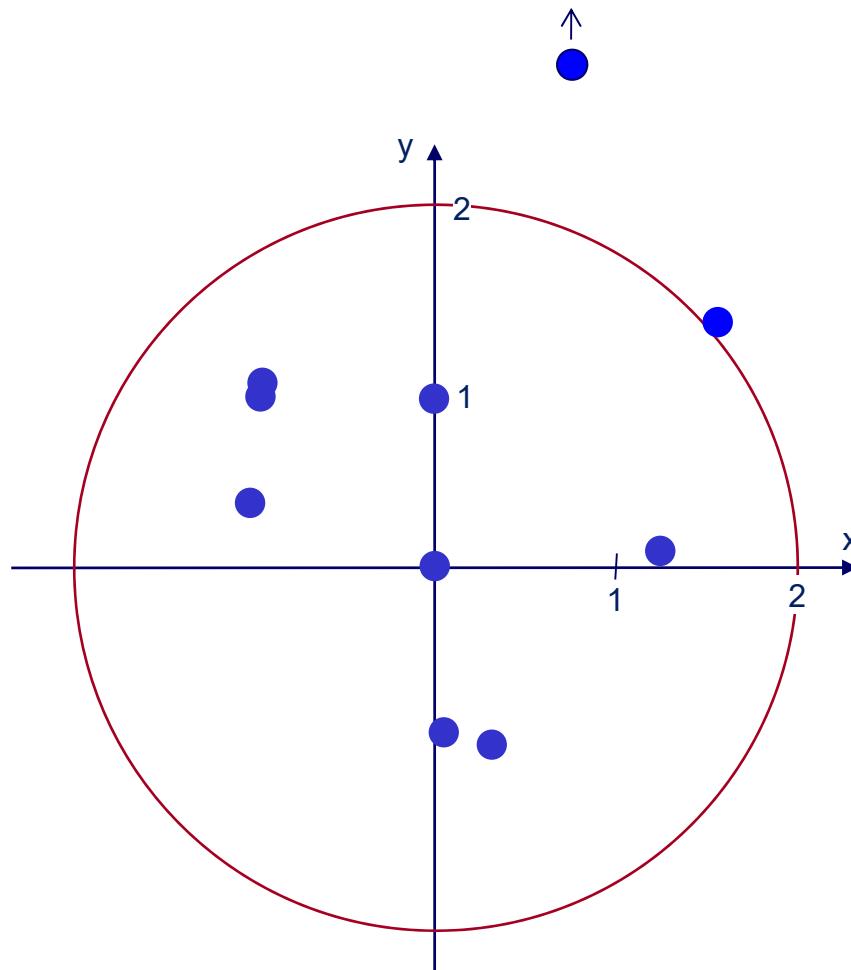
$n \geq 0$

$$\begin{aligned}x_{n+1} &= x_n^2 - y_n^2 + 0 \\y_{n+1} &= 2 \times x_n \times y_n + 1\end{aligned}$$

$$\begin{aligned}(x_0, y_0) &= (0, 0) \\(x_1, y_1) &= (0, 1) \\(x_2, y_2) &= (-1, 1) \\(x_3, y_3) &= (0, -1) \\(x_4, y_4) &= (-1, 1) \\(x_5, y_5) &= (0, -1) \\(x_6, y_6) &= (-1, 1) \\&\dots\end{aligned}$$

- Afstanden fra $(0,0)$ går ikke mod uendelig
 - Punktet $(0,1)$ tilhører Mandelbrotmængden

Følgen for $(a,b) = (0, 1.01)$



$n = 0$

$$\begin{aligned}x_0 &= 0 \\y_0 &= 0\end{aligned}$$

$n \geq 0$

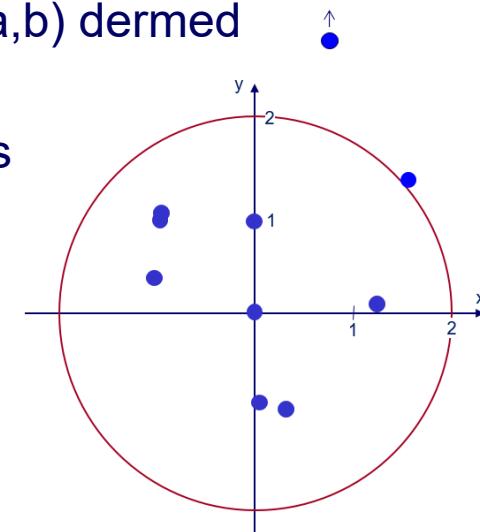
$$\begin{aligned}x_{n+1} &= x_n^2 - y_n^2 + 0 \\y_{n+1} &= 2 \times x_n \times y_n + 1.01\end{aligned}$$

$$\begin{aligned}(x_0, y_0) &= (0, 0) \\(x_1, y_1) &= (0, 1.01) \\(x_2, y_2) &= (-1.02, 1.01) \\(x_3, y_3) &= (0.02, -1.05) \\(x_4, y_4) &= (-1.10, 0.97) \\(x_5, y_5) &= (0.28, -1.12) \\(x_6, y_6) &= (-1.18, 0.38) \\(x_7, y_7) &= (1.26, 0.12) \\(x_8, y_8) &= (1.57, 1.32) \\(x_9, y_9) &= (0.72, 5.14) \\&\dots\end{aligned}$$

- Afstanden fra $(0,0)$ går mod uendelig
 - Punktet $(0,1.01)$ tilhører ikke Mandelbrotmængden

Beregning af Mandelbrotmængden

- Man kan vise, at hvis talfølgen en gang kommer uden for cirklen (med radius 2) vil afstanden til (0,0) gå mod uendelig
 - Når det sker, kan vi stoppe beregningen og konkludere, at punktet ikke tilhører Mandelbrotmængden
- For at lave en præcis beregning af Mandelbrot mængden skal vi beregne uendelige talfølger
 - Det kan man ikke gøre i endelig tid
 - Derfor vælger vi et forudbestemt, maksimalt antal iterationer **max**
 - Hvis talfølgen **ikke** kommer uden for cirklen indenfor **max** iterationer, antager vi, at den **aldrig** kommer udenfor, og at (a,b) dermed tilhører mandelbrotmængden
 - Jo større vi sætter **max**, jo mere præcist beregnes Mandelbrotmængden (og jo mere tid skal der bruges på beregningen)

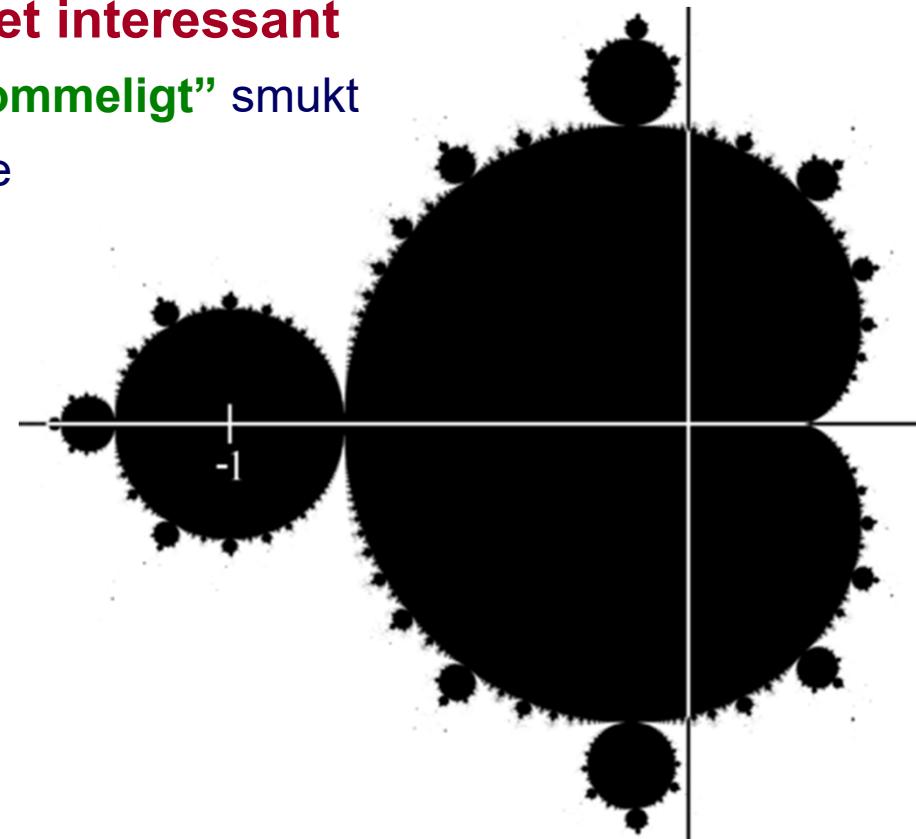


Sort/hvid algoritme (pseudokode)

```
for each point (a,b) {  
    n = 0;  
    (x,y) = (0,0);  
  
    while(|(x,y)| ≤ 2 && n < max) {  
        beregn næste punkt i talfølgen;  
        n++;  
    }  
  
    if(n = max) {  
        color(a,b) = black; // Member  
    }  
    else {  
        color(a,b) = white; // Not member  
    }  
}
```

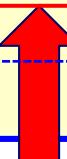
Mandelbrotmængdens egenskaber

- Mandelbrotmængden har en "flosset" kant
 - nogle punkter er oplagt med
 - og nogle punkter er oplagt ikke med
- Langs kanten sker der noget interessant
 - der på samme tid er "guddommeligt" smukt
 - og "djævelsk" svært at fatte
- Det vil vi om et øjeblik studere nærmere

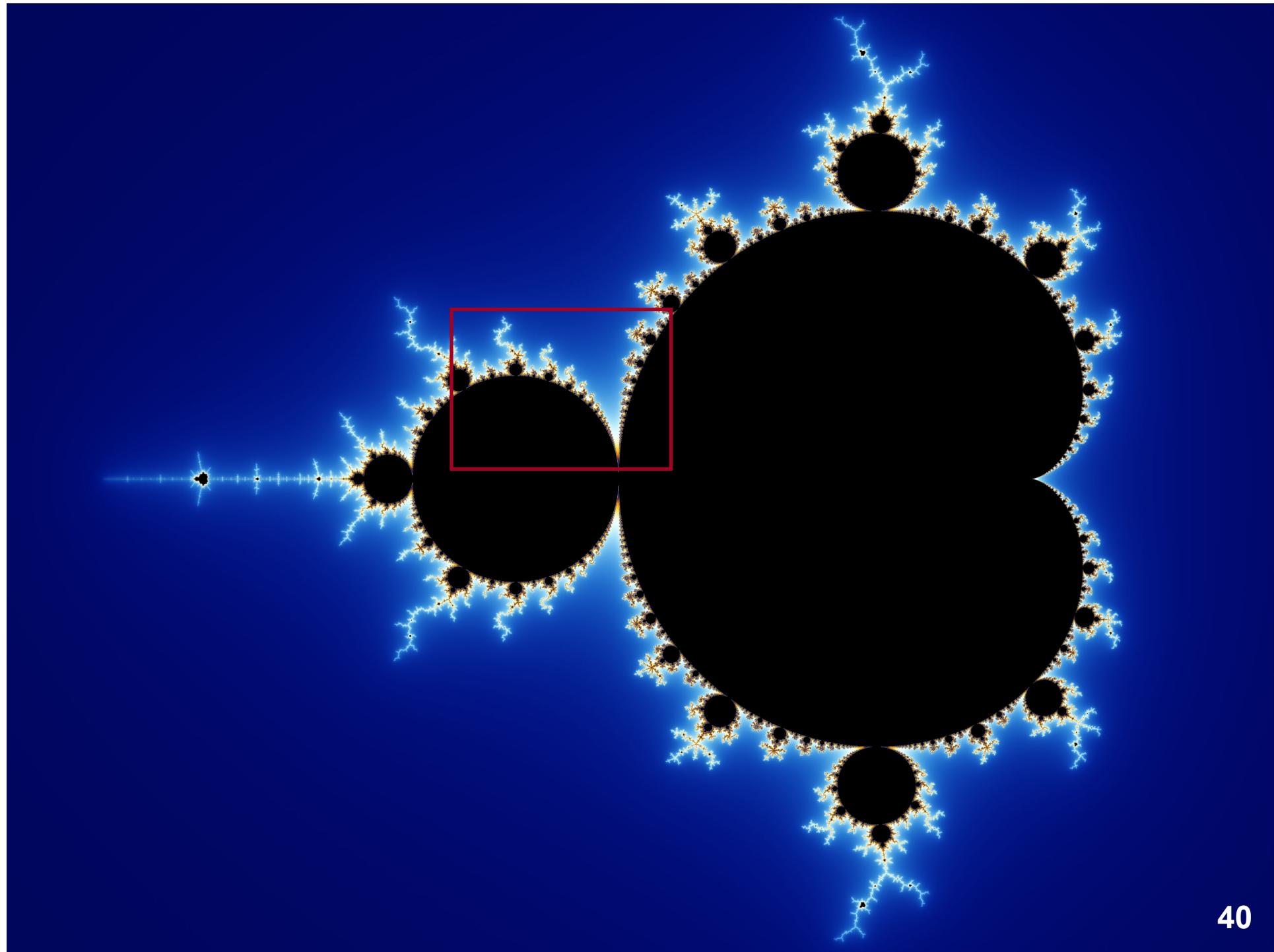


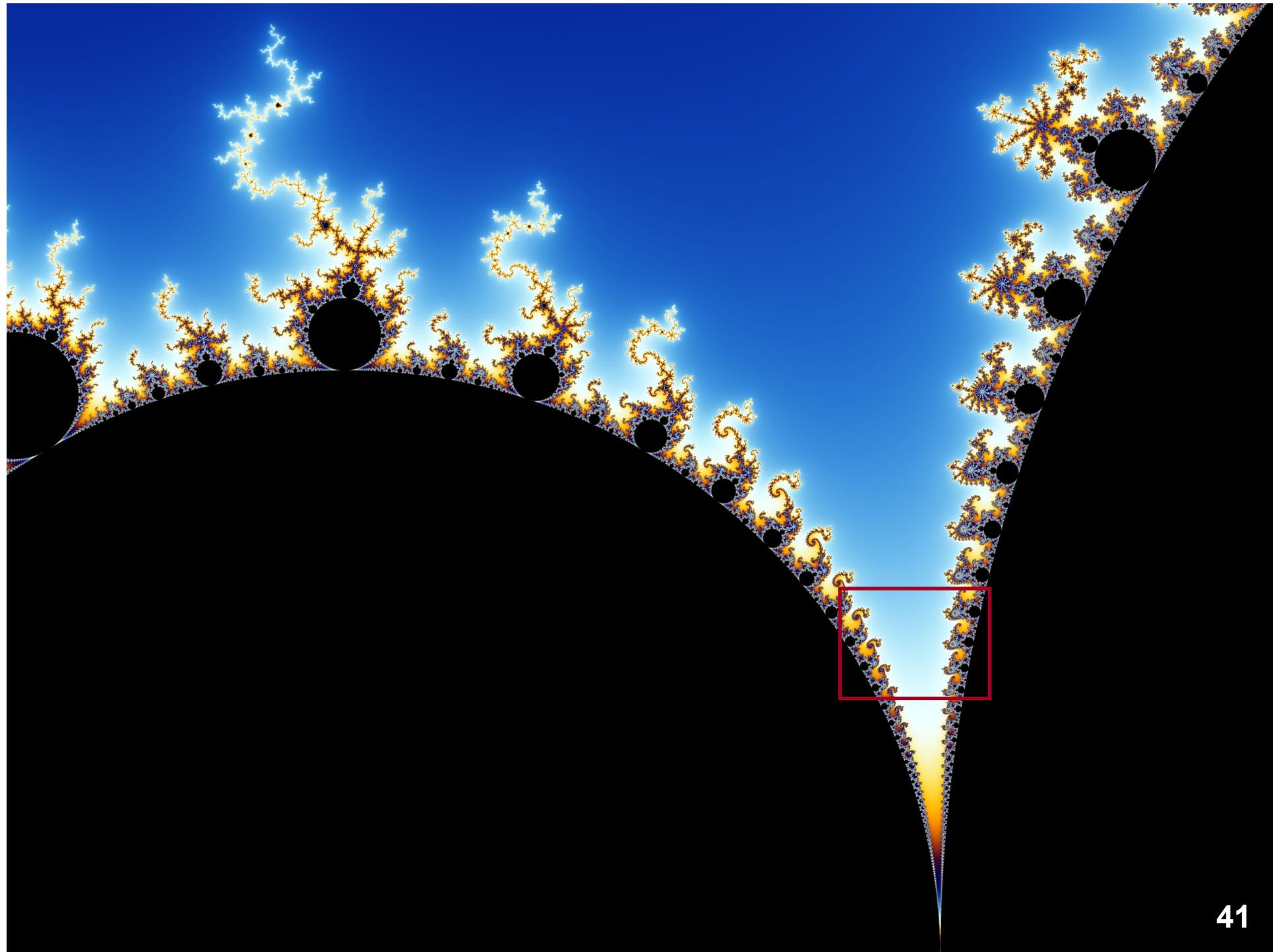
Algoritme med farver

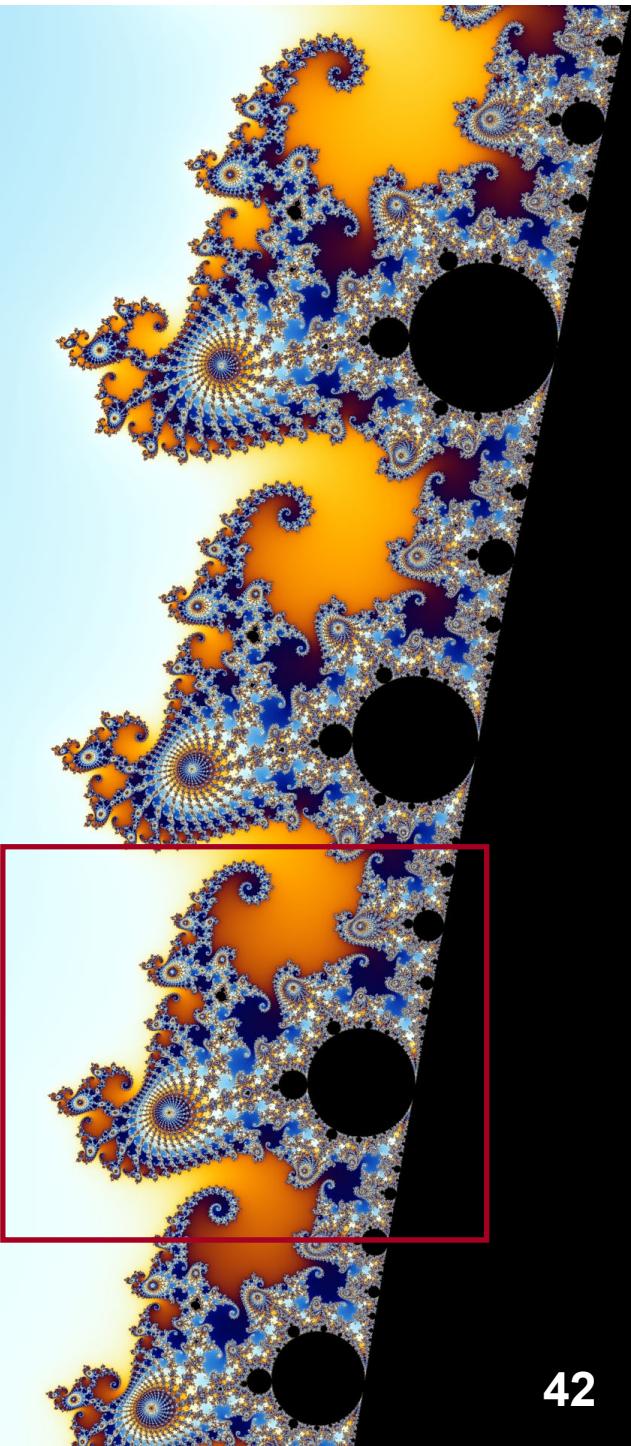
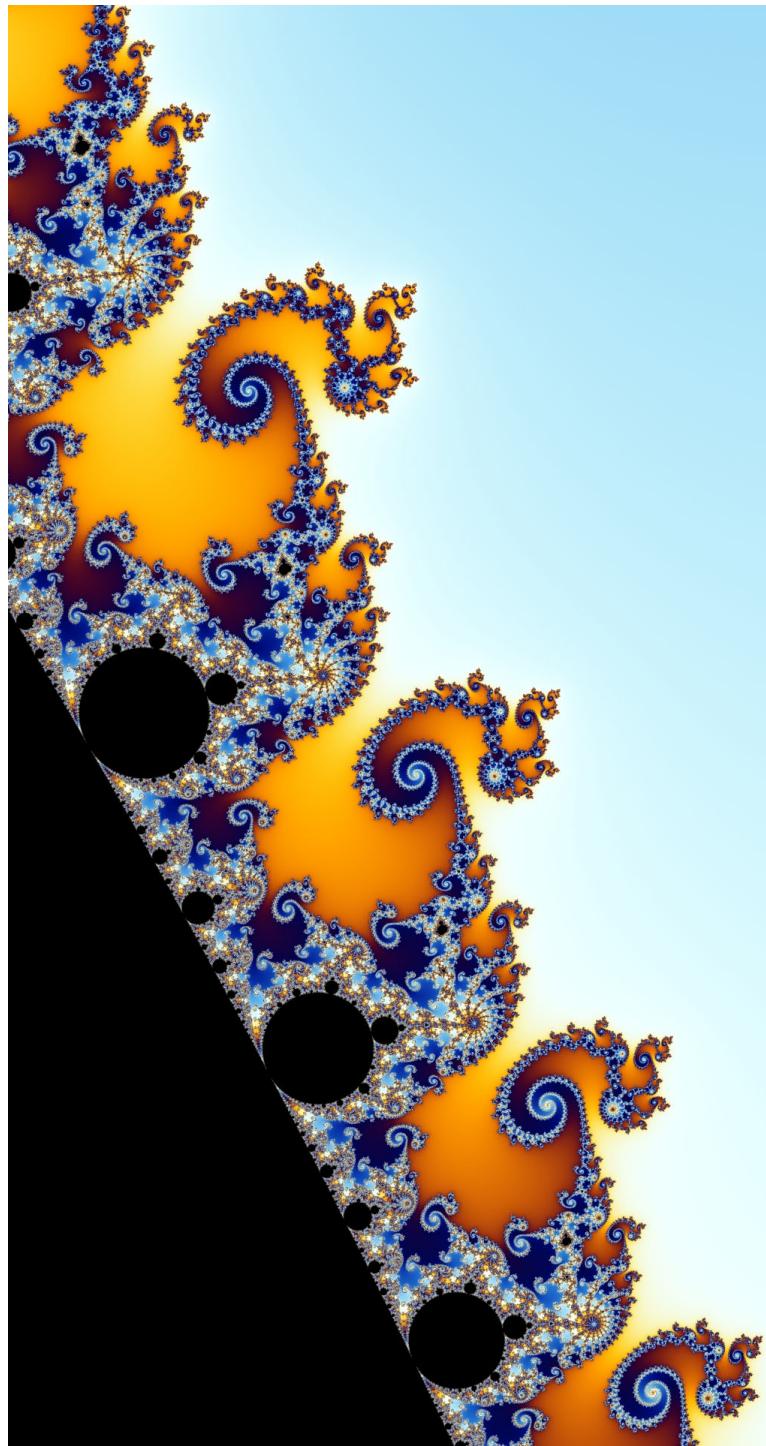
```
for each point (a,b) {  
    n = 0;  
    (x,y) = (0,0);  
  
    while(|(x,y)| ≤ 2 && n < max) {  
        beregn næste punkt i talfølgen;  
        n++;  
    }  
  
    if(n = max) {  
        color(a,b) = black;      // Member  
    }  
    else {  
        color(a,b) = color(n); // Not member  
    }  
}
```

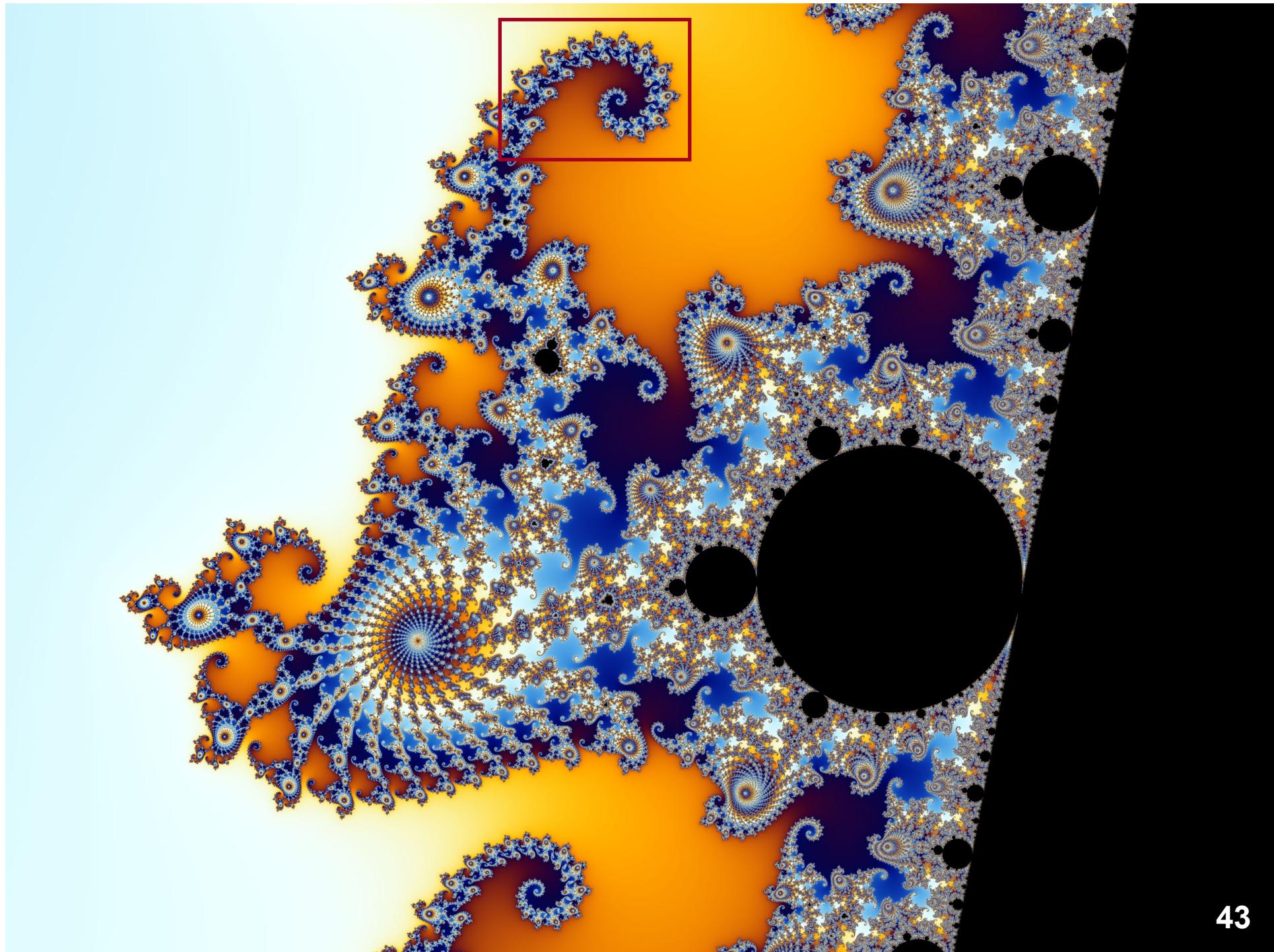


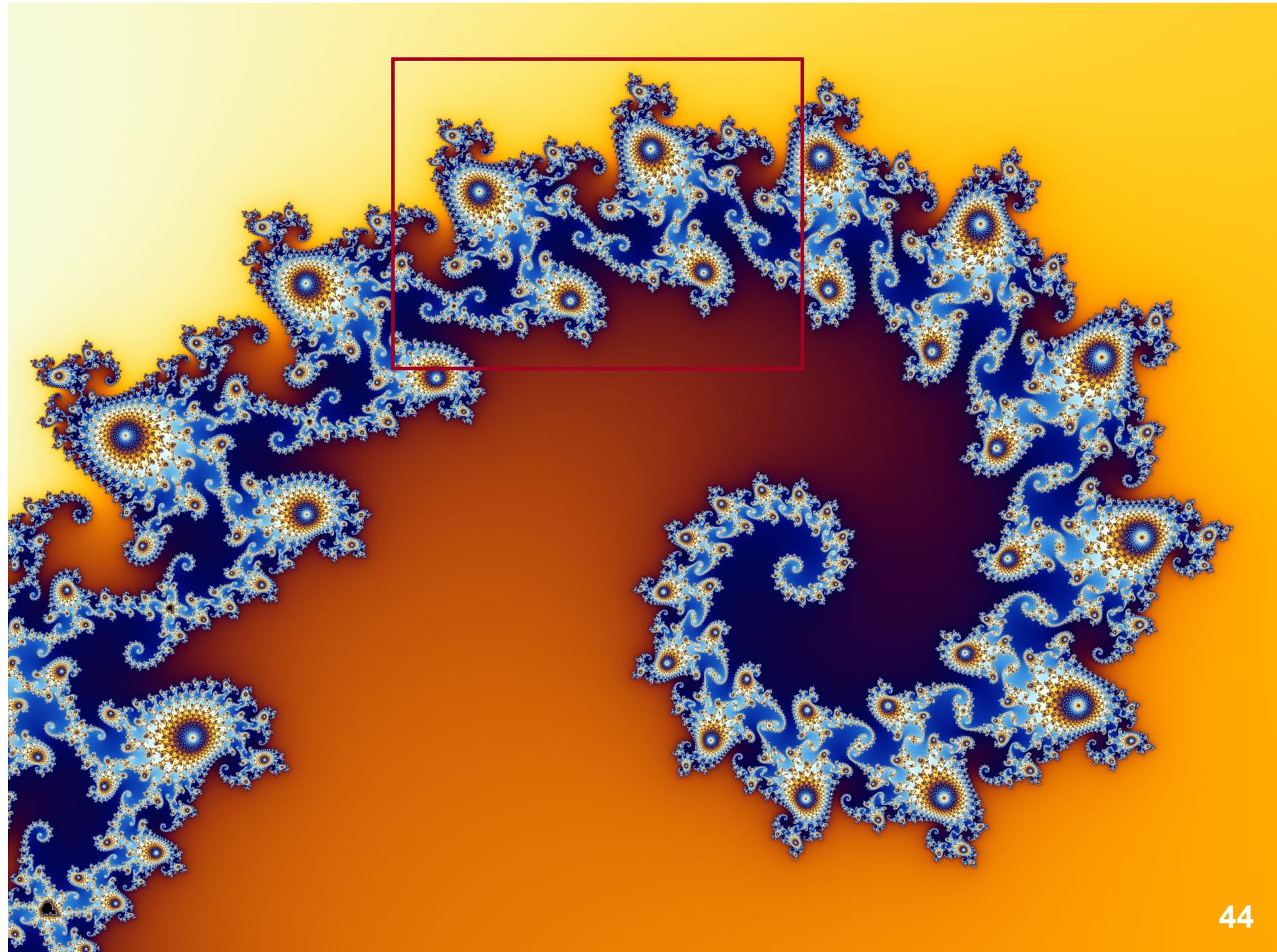
For punkter udenfor Mandelbrotmængden bestemmes farven nu af antallet skridt inden vi kommer uden for cirklen (via en ikke-trivial funktion color)

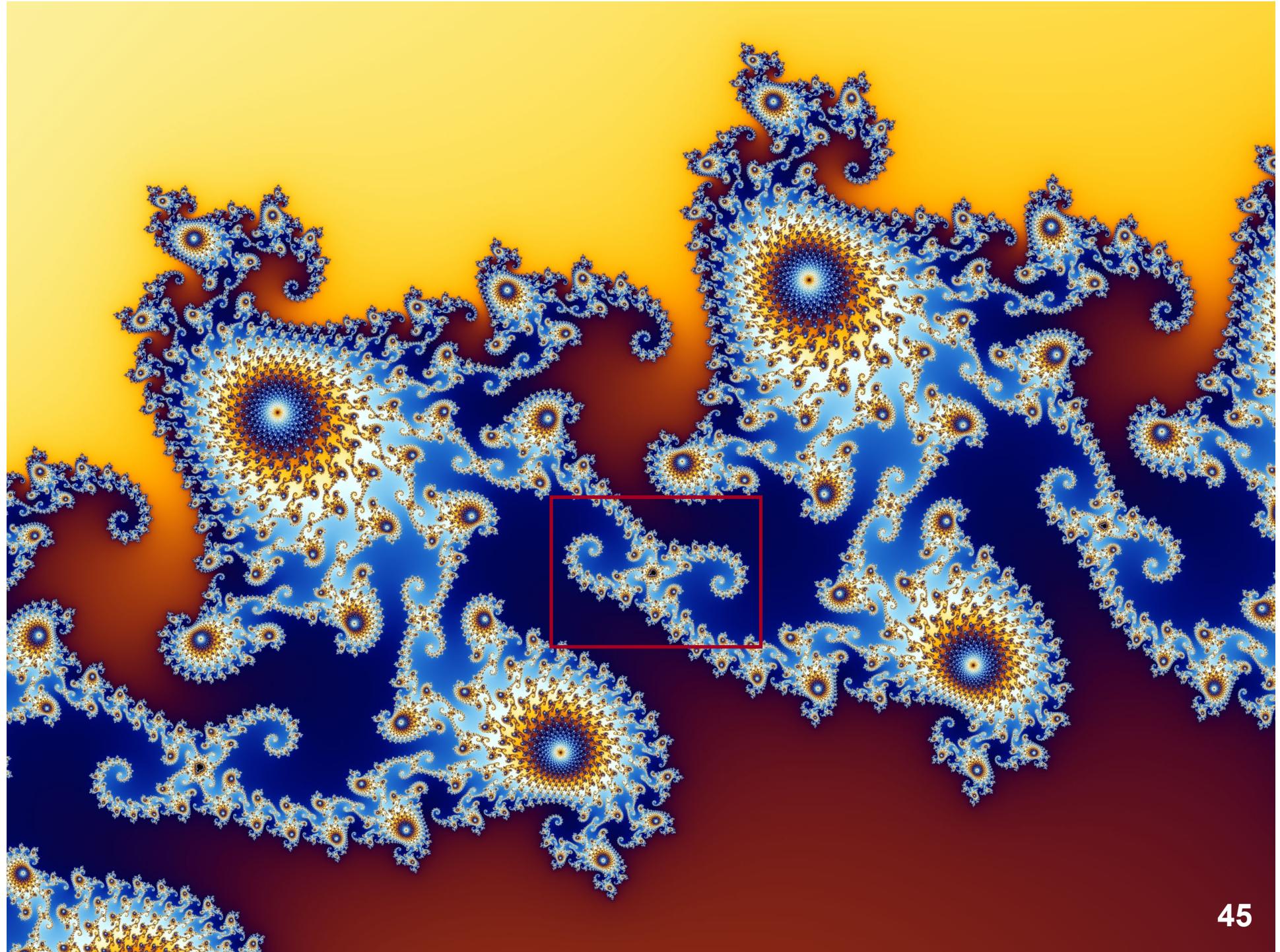


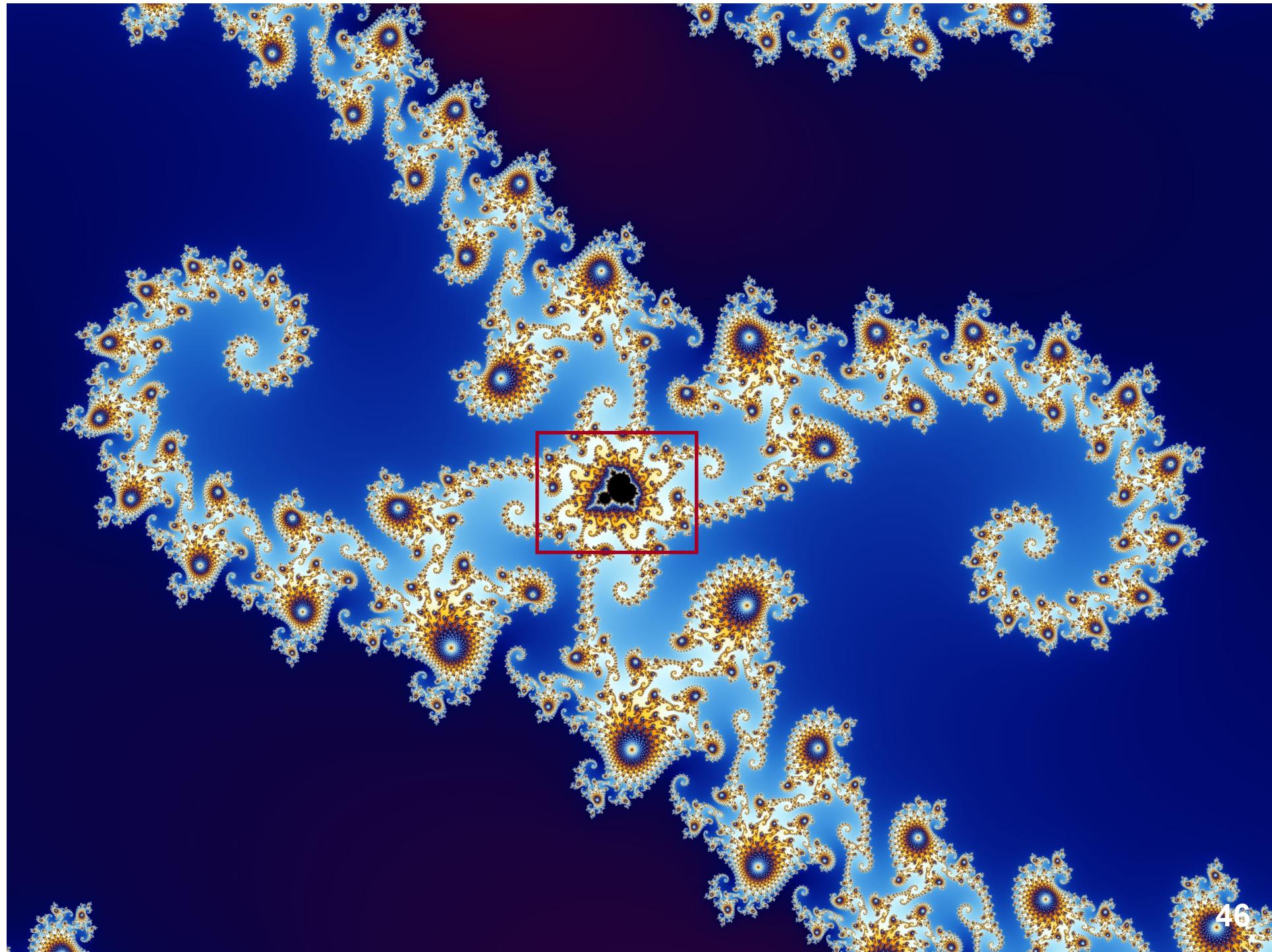


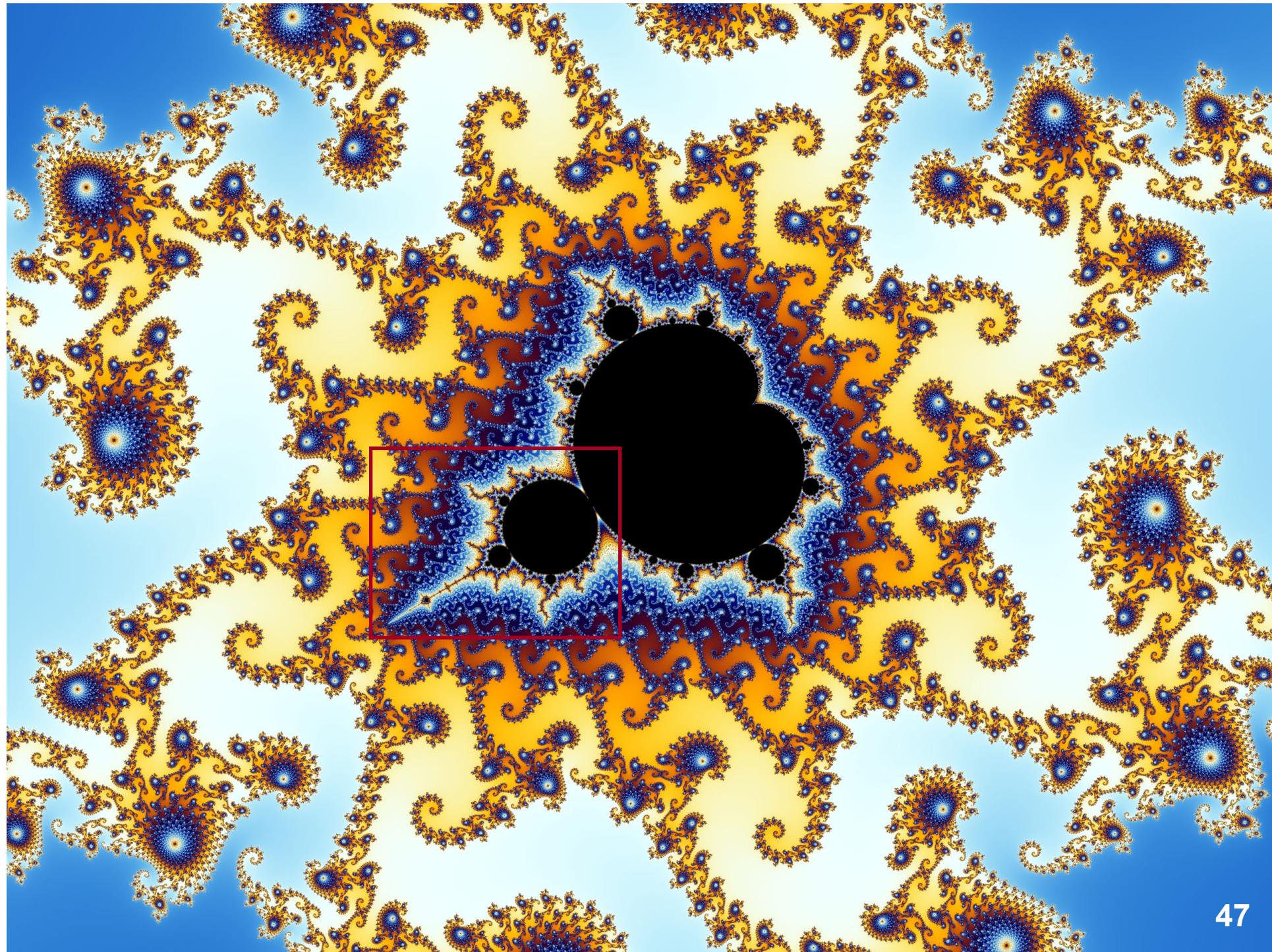


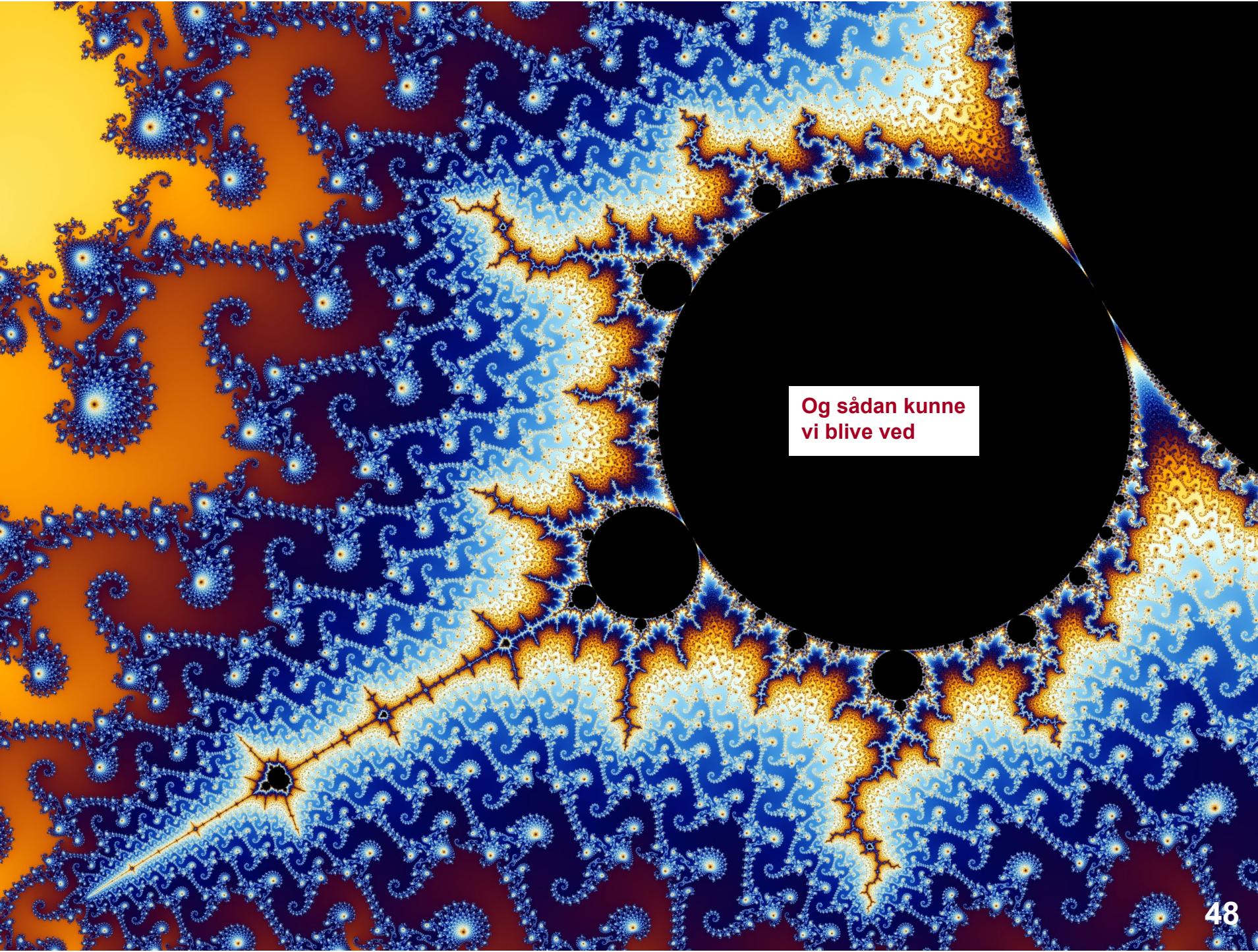






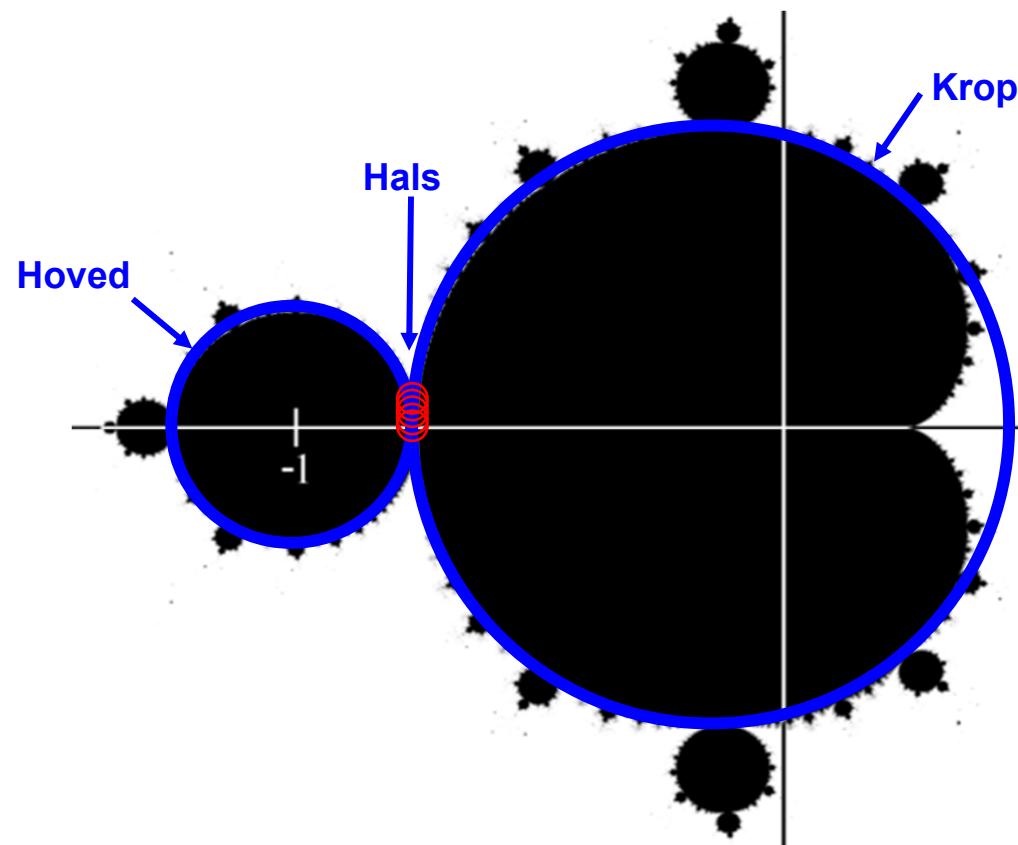






Og sådan kunne
vi blive ved

Til sidst vil vi undersøge "halsen"



ε	Iterationer
0.1	33
0.01	315
0.001	3143
0.0001	31417
0.00001	314160
0.000001	3141593
0.0000001	31415927

π

3.14159265358979...

- **Vi vil vise, at der ikke er nogen hals mellem hovedet og kroppen**
 - Lad os betragte $(x, y) = (-0.75, \varepsilon)$ for ε gående mod 0
 - Det viser sig, at ingen af dem tilhører Mandelbrotmængden, hvilket betyder, at der er hvidt helt ind til x-aksen



Tak for nu!

Held og lykke
til eksamen!