Inlämningsuppgifter

I inlämningsuppgifterna ska du lära dig att utveckla program som löser givna problem och att skriva rapporter som beskriver dina lösningar. I detta kapitel finns de båda uppgifterna.

Regler:

- Uppgifterna är obligatoriska. Det innebär att du måste bli godkänd på uppgifterna under kursens gång.
- Uppgifterna ska lösas i grupper om två personer. Ni ska själva bilda grupper. Regler för samarbete finns i bilaga A, sidan 69.
- Uppgifterna ska redovisas med skriftliga rapporter och med program. Närmare anvisningar finns i bilaga B, sidan 71.
- Tider för inlämning av inlämningsuppgifterna finns i kursprogrammet. Det är viktigt att ni lämnar in i tid. Ni kan få dispens om ni har särskilda skäl, men då måste ni begära detta hos kursledaren i god tid före inlämningsdatum. Om ni inte lämnar in i tid och inte har fått dispens så får ni inte göra uppgiften förrän nästa gång kursen går, dvs nästa år, och då får ni inte något slutbetyg i kursen i år.
- Ni kan bli kallade till muntlig redovisning av en inlämningsuppgift. Då måste ni kunna redogöra i detalj för era lösningar.

Varje inlämningsuppgift ska redovisas med en skriftlig rapport och med de Javaprogram som ni har skrivit. Gör följande:

- Skriv ut rapporten, häfta ihop den, lämna den i det grå skåpet i foajén utanför institutionen.
- Arkivera projektkatalogen i Eclipse (Export > General > Archive File).
- Skriv ett e-brev till `eda016@cs.lth.se`, bifoga det arkiverade projektet (*.zip*-filen). Ämnesraden i brevet ska se ut så här:

  ```
  uppgiftens-namn by id1 id2
  ```

  Uppgiftens namn är `turtlerace` (uppgift 1) eller `mandelbrot` (uppgift 2). `by` är en avgränsare. `id1` och `id2` är era datoridentiteter, till exempel `dat14xyn` eller `dic14yka`. Skriv era namn i klartext i brevet.

Vi rättar rapporterna så snart vi hinner, normalt inom en arbetsvecka räknat från sista inlämningsdag.

- Om uppgiften blir godkänd får du meddelande om det via e-post, och du kan senare hämta din godkända rapport under en föreläsning.
- Om uppgiften inte blir godkänd får du meddelande om det via e-post, och du ska hämta rapporten hos kurssekreteraren. Du ska *inom en vecka* lämna in en förbättrad version. Du

måste bifoga den ej godkända rapporten, så att vi inte behöver börja om från början med rättningen. I din förbättrade rapport ska du naturligtvis ha åtgärdat alla brister som rättaren påpekat.

Inlämningsuppgift 1 — TurtleRace

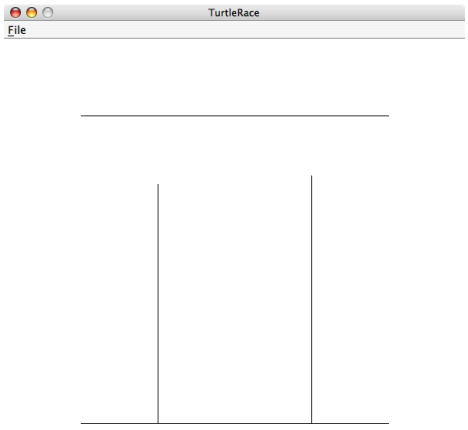
Mål: I denna uppgift ska du lära dig att välja operationer till givna klasser. Du ska också lära dig att implementera klasser som du själv har specificerat.

Bakgrund

En kapplöpningstävling mellan två deltagare äger rum på en kapplöpningsbana. Banan har ett fixt utseende, till exempel en bestämd längd. Flera lopp kan avgöras på samma bana.

Uppgift

Ett program som simulerar ett lopp mellan två sköldpaddor på en kapplöpningsbana ska skrivas. Kapplöpningsbanan består av två banor, en för varje sköldpadda. Båda banorna har en gemensam startlinje (underst i figuren nedan) och en gemensam mållinje (överst i figuren). I figuren syns startlinjen, mållinjen och två tävlande sköldpaddor. Den högra sköldpaddan har hunnit något längre mot målet än den vänstra.



Om uppgiften vore att rita fyra streck i ett ritfönster så skulle det bara behövas några rader i en `main`-metod för att lösa den. Men nu är uppgiften att skriva ett simuleringsprogram, och ett sådant ska efterlikna det verkliga systemet så långt som möjligt. I programmet ska det alltså finnas klasser som representerar kapplöpningsbanan, sköldpaddorna och själva loppet.

Klasserna ska ha följande specifikationer:

Huvudprogrammet: `TurtleRace`

```
/** Skapar sköldpaddorna, kapplöpningsbanan och ett lopp,
    genomför loppet */
static void main(String[] args);
```

En kapplöpningsbana: `RaceTrack`

```
/** Skapar en kapplöpningsbana. yStart och yFinish är
    y-koordinaterna för start- och mållinje */
RaceTrack(int yStart, int yFinish);

/** Ritar kapplöpningsbanan i fönstret w */
void draw(SimpleWindow w);

// ... infoga flera metoder här
```

Ett lopp: `RacingEvent`

```
/** Skapar ett lopp mellan sköldpaddorna t1 och t2 på
    kapplöpningsbanan track */
RacingEvent(RaceTrack track, Turtle t1, Turtle t2);

// ... infoga metoder här
```

En sköldpadda: `Turtle`

```
// ... enligt laboration 4
```

Klassen `Turtle` är den klass som du skrev i datorlaboration 4. Detta är ett exempel på att klasser kan återanvändas — du ska inte skriva en specialversion av `Turtle` för denna uppgift.

I klassen `RacingEvent` ska det finnas en metod som genomför ett lopp mellan sköldpaddorna (man ska alltså *inte* genomföra loppet i konstruktorn). I metoden ska sköldpaddorna ställas upp på startlinjen och gå framåt tills en av dem nått mållinjen (tänk på att sköldpaddorna vandrar uppåt i fönstret dvs mot mindre y-värden). Låt sköldpaddorna omväxlande ta steg av slumpmässig längd i intervallet $[0, 2]$. Lägg in en fördröjning när båda sköldpaddorna tagit ett steg, så att man ser hur sköldpaddorna rör sig.

Välj själv lämpliga operationer till klassen `RaceTrack`. Du bör när du skrev metoden i klassen `RacingEvent` ha upptäckt vilka dessa operationer är. `RaceTrack` ska inte känna till att det är sköldpaddor som tävlar. Om man till exempel vill skriva en klass `Dog` och låta två hundar tävla ska man inte behöva ändra kapplöpningsbanan.

Skriv också klassen `TurtleRace` med `main`-metoden, som skapar de objekt som behövs och som genomför ett lopp. Påbörja inte loppet förrän användaren har tryckt på en musknapp.

Programskrivning

Skapa ett Eclipse-projekt *inl1* (följ instruktionerna i uppgift 1–3, datorlaboration 5). Filerna med klasserna ska ha följande uppbyggnad:

RaceTrack.java

```
import se.lth.cs.window.SimpleWindow;
public class RaceTrack { ... }
```

RacingEvent.java

```
import java.util.Random;
import se.lth.cs.window.SimpleWindow;
import lab4.Turtle;
public class RacingEvent { ... }
```

Turtle.java

```
import se.lth.cs.window.SimpleWindow;
public class Turtle { ... }
```

TurtleRace.java

```
import se.lth.cs.window.SimpleWindow;
import lab4.Turtle;
public class TurtleRace {
    public static void main(String[] args) {
        ...
    }
}
```

Utökningar av programmet — inte obligatoriskt

Om du har lust och tid får du gärna förbättra din lösning, till exempel enligt följande:

- Rita en mera avancerad kapplöpningsbana, till exempel med text som visar vilken av linjerna som är start- respektive mållinje och med nummer på banorna.
- Skriv ut vilken av sköldpaddorna som vann loppet.
- Låt sköldpaddorna vandra rakt fram men med snedsteg som gör att de ibland avviker från den raka vägen. Man måste då kontrollera så att inte någon sköldpadda hamnar utanför sin bana.
- Gör det möjligt att välja att köra loppet med fler än två sköldpaddor.

Om du gör utökningar så bör du tänka på att det är samma kvalitetskrav på utökningarna som på de obligatoriska delarna. I rapporten måste du tydligt beskriva vilka ändringar som du har gjort.

Rapport

Följ anvisningarna i bilaga B. Alla program som du själv har skrivit ska vara med i rapporten, alltså även klassen `Turtle`.

Inlämningsuppgift 2 — Mandelbrot

Mål: I denna uppgift ska du lära dig att implementera komplicerade algoritmer med numeriska beräkningar.

Bakgrund

Den fransk-amerikanske matematikern Benoît Mandelbrot (född i Polen) införde år 1975 ett nytt matematiskt begrepp, fraktal. Fraktaler har anknytning till den relativt unga matematiska vetenskapen kaosteori. En fraktal är en geometrisk figur med ovanliga egenskaper, till exempel är den sådan att mönster i figuren upprepas i det oändliga när man förstorar olika delar av den.

Ett välkänt exempel på en fraktal är Mandelbrotmängden M, som är en delmängd av de komplexa talen C. Den innehåller ett oändligt antal punkter men har en begränsad utsträckning i det komplexa talplanet. Man kan undersöka om ett komplext tal c tillhör M på följande sätt:

- Definiera en talföljd (Mandelbrotföljden):

$$z_k = \begin{cases} 0, & k = 0 \\ z_{k-1}^2 + c, & k = 1, 2, \ldots \end{cases}$$

- z_0 är 0, så z_1 blir c. Nästa z-värde blir $z_2 = c^2 + c$, nästa igen blir $z_3 = c^4 + 2c^3 + c^2 + c$, och så vidare.
- Följden $z_0, z_1, z_2, \ldots$ kan bete sig på tre sätt:
  1. Den kan konvergera mot en punkt i det komplexa talplanet. Till exempel ger $c = i/2$ en följd som konvergerar mot $-0.136009 + 0.393075\ i$.
  2. Den kan ”konvergera” mot två eller flera punkter som upprepas periodiskt. Till exempel ger $c = i$ en följd som omväxlande antar värdena $-1 + i$ och $-i$.
  3. Den kan divergera mot oändligheten. Detta gäller till exempel för $c = 2$.

Definition: Ett komplext tal c tillhör Mandelbrotmängden M om följden av komplexa tal *inte* divergerar, det vill säga om något av fallen 1 eller 2 ovan gäller.

Om man i ett datorprogram ska undersöka om en punkt tillhör Mandelbrotmängden så kan man inte använda definitionen direkt — för att kunna avgöra om en följd konvergerar eller divergerar skulle ju man behöva iterera oändligt länge. I ett program nöjer man sig därför med att göra ett begränsat antal iterationer och därefter använda följande sats:

Om det för något k gäller att $\lvert z_k \rvert > 2$ så divergerar Mandelbrotföljden.

Uppgift

Du ska skriva ett program som beräknar Mandelbrotmängden och åskådliggör punkterna i mängden i ett komplext talplan. Programmets användargränssnitt är färdigskrivet. Det är en god idé att följa den arbetsordning som beskrivs nedan.

Börja med att skapa ett projekt *inl2* för filerna i uppgiften.

1 Komplexa tal

I uppgiften behövs en klass som beskriver komplexa variabler. Du behöver bara implementera de operationer på komplexa tal som kommer att behövas i detta program. Skriv alltså en klass `Complex` med följande specifikation:

```
/** Skapar en komplex variabel med realdelen re och imaginärdelen im */
Complex(double re, double im);

/** Tar reda på realdelen */
double getRe();

/** Tar reda på imaginärdelen */
double getIm();

/** Tar reda på talets absolutbelopp i kvadrat */
double getAbs2();

/** Adderar det komplexa talet c till detta tal */
void add(Complex c);

/** Multiplicerar detta tal med det komplexa talet c */
void mul(Complex c);
```

Skriv ett testprogram som kontrollerar att din `Complex`-klass fungerar. Metoderna `getRe` och `getIm` kommer du bara att använda i testprogrammet och när du ritar en cirkel (moment 3). Anledningen till att klassen bara har en operation för att beräkna absolutbeloppet i kvadrat, inte för absolutbeloppet, är att man vill slippa en tidsödande kvadratrotsberäkning.

Eftersom programmet vid varje exekvering kommer att göra flera miljoner beräkningar med `Complex`-objekt är det viktigt att räkneoperationerna implementeras effektivt. Detta innebär att operationerna ska implementeras i rektangulära koordinater, som inte kräver några tidsödande beräkningar av trigonometriska funktioner.

Var försiktig när du implementerar operationen `mul` så att du under beräkningarna inte råkar förstöra ett värde som du senare behöver. Tänk särskilt på fallet `z.mul(z)`.

2 Orientering om användargränssnittet

När du senare har genererat Mandelbrotmängden ska du visa en bild av mängden i ett fönster. Användargränssnittet är färdigskrivet (klassen `se.lth.cs.ptdc.fractal.MandelbrotGUI`). En länk till specifikationen av klassen finns på kursens hemsida. Studera specifikationen noggrant medan du läser vidare. Börja med att skriva en klass `Mandelbrot` med en `main`-metod som skapar ett objekt av `MandelbrotGUI` (använd konstruktorn utan parametrar). Kompilera och testa!

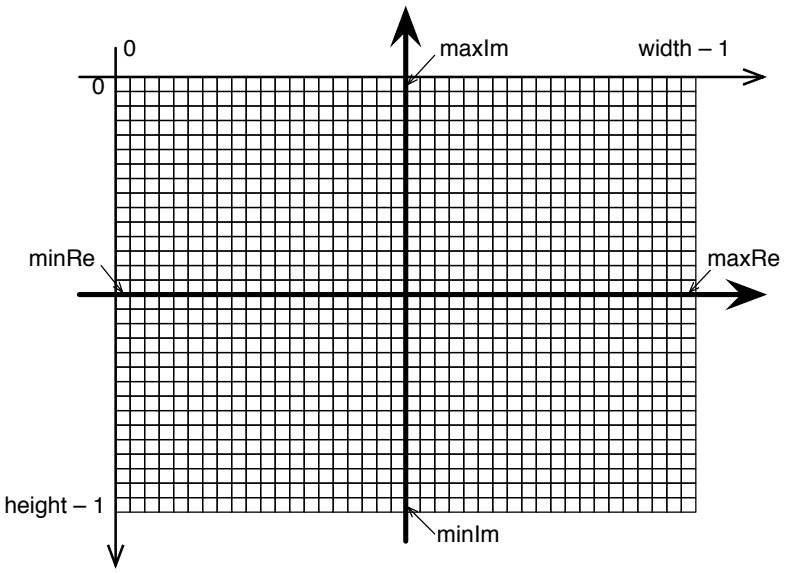
I fönstret visas ett komplext talplan och fyra rutor som visar intervallen på planets koordinataxlar. Man kan förstora (”zooma in”) en del av planet genom att med musen dra ut en rektangel. Testa att detta fungerar, dvs att min- och maxvärdena på koordinataxlarna ändras när du zoomar.

Överst i fönstret finns några knappar och menyer. Som du märker fungerar de ännu inte (avsikten är förstås att du ska implementera deras funktion). Exempelvis går det inte att avsluta programmet med Quit-knappen.

Det finns också en textruta märkt Extra där du kan mata in vilka data du vill och hämta dem till programmet. Se avsnitt 7.

Din `main`-metod ska kommunicera med användargränssnittet genom att fråga efter kommandon som användaren ger genom att trycka på knapparna. I `MandelbrotGUI` finns en metod `getCommand` som returnerar ett heltal som är numret på ett kommando. Vidare finns ett antal publika konstanter som är numren på kommandona (`RENDER`, `RESET`, osv).

Lägg i `main`-metoden in en repetitionssats där du frågar efter ett kommando och därefter gör olika saker beroende vad som returneras. Använd en `switch`-sats — förbered med `case`-grenar för varje kommando men gör tills vidare bara en utskrift som identifierar kommandot. Testa! Ta bort utskrifterna efterhand som du implementerar de olika kommandona.



Figur 1: Ritytans koordinatsystem och det komplexa koordinatsystemet.

Implementera slutligen kommandona `RESET` och `QUIT`. `RESET` ska medföra att axlarnas min- och maxvärden återställs till de ursprungliga och att ritytan töms. `QUIT` ska medföra att programmet avslutas med `System.exit(0)`.

Kontrollera att du nu kan avsluta programmet genom att trycka på `QUIT`-knappen. Testa också `RESET`-funktionen.

3 Rita en cirkel (det svåraste problemet)

I denna deluppgift ska du börja skriva en klass `Generator` som ska beräkna punkterna i Mandelbrotmängden och visa dem i fönstret. Det gäller att för varje punkt c i det komplexa talplanet (den del som syns i fönstret) avgöra om punkten tillhör Mandelbrotmängden eller inte. Om c tillhör mängden färgas motsvarande pixel svart (`java.awt.Color.BLACK`), annars färgas pixeln vit (`java.awt.Color.WHITE`).

Till att börja med så nöjer vi oss dock med att rita enklare bilder än fraktaler, för att vi ska få ordning på de olika koordinatsystem som är inblandade i problemet. Man ska också kunna rita bilder med olika upplösning; det väntar vi också med.

I uppgiften har vi att göra med två koordinatsystem. Det första systemet är det komplexa talplanet. I ritfönstret visas en del av detta: från början visas intervallet $[-2.4, 0.8]$ av den reella axeln och intervallet $[-1.4, 1.4]$ av den imaginära axeln. Användaren kan ändra intervallen genom att zooma med musen. Det finns operationer i `MandelbrotGUI` för att ta reda på intervallgränserna: `getMinimumReal`, `getMaximumReal`, `getMinimumImag` och `getMaximumImag`.

Det andra koordinatsystemet är ritytans system (detta kallas i fortsättningen för ritsystemet). Detta system har som vanligt en bredd och en höjd räknat i pixlar. Det finns operationer för att ta reda på dessa värden (`getWidth` och `getHeight`). Ritsystemet har origo i övre vänstra hörnet av ritytan och y-axeln är riktad nedåt.

Varje pixel i ritsystemet motsvaras alltså av ett komplext tal. Till exempel motsvaras punkten med index $0, 0$ i ritsystemet av ett komplext tal med realdelen `getMinimumReal()` och imaginärdelen `getMaximumImag()`.

I figur 1 visas ritytan. Den har bredden `width` pixlar och höjden `height` pixlar. `minRe` och `maxRe` är den reella axelns gränser, `minIm` och `maxIm` är den imaginära axelns gränser. Lägg märke till att y-axeln i de båda koordinatsystemen pekar i olika riktningar.

Klassen `Generator` ska ha följande metoder:

```
/** Ritar en bild i fönstret i användargränssnittet gui */
public void render(MandelbrotGUI gui);

/** Skapar en matris där varje element är ett komplext tal som
    har rätt koordinater (se beskrivning nedan) */
private Complex[][] mesh(double minRe, double maxRe,
                         double minIm, double maxIm,
                         int width, int height);
```

Anvisningar för `mesh`. Matrisen som returneras ska ha storleken `height` rader och `width` kolonner och ska användas för att till varje bildelement associera rätt komplext tal. Tänk dig alltså att matrisen läggs ovanpå ritytan.

Fyll i rätt komplext tal i varje element i matrisen. Elementet med index `[0][0]` ska ha realdelen `minRe` och imaginärdelen `maxIm`, elementet med index `[height - 1][width - 1]` ska ha realdelen `maxRe` och imaginärdelen `minIm`. Kontrollera dina formler så att de stämmer.

Anvisningar för `render`. I denna metod ska du göra följande:

1. Anropa `disableInput` i `MandelbrotGUI`. Anropet medför att knapparna i användargränssnittet inte reagerar på tryck och att beräkningarna inte störs av att användaren gör operationer i fönstret.
2. Anropa `mesh` för att skapa matrisen med komplexa tal. Kalla matrisen `complex`.[1]
3. Det behövs ytterligare en matris som ska innehålla bildpunkterna som ritas. Deklarera matrisen med `Color[][] picture`. Matrisen ska vara lika stor som ritytan (detta kommer att ändras senare, när du ska rita bilder med olika upplösning).
4. Gå igenom matrisen `picture` rad för rad och fyll i en färg på varje plats. Färgen väljs enligt följande: om `complex[i][k]` har absolutbeloppet > 1 så sätts `picture[i][k]` till `Color.WHITE`, annars väljs en unik färg för varje kvadrant av planet. Använd till exempel `Color.RED` till den första kvadranten, `Color.BLUE` till den andra, osv. Resultatet ska bli att en cirkel med fyra färger ritas.
5. När varje element har fått en färg ska bilden ritas med `putData` i `MandelbrotGUI`.
6. Anropa till sist `enableInput` för att återställa funktionen hos användargränssnittets knappar.

Komplettera sedan din `main`-metod genom att implementera kommandot `RENDER` i `switch`-satsen med ett anrop till `render`-metoden. Testa!

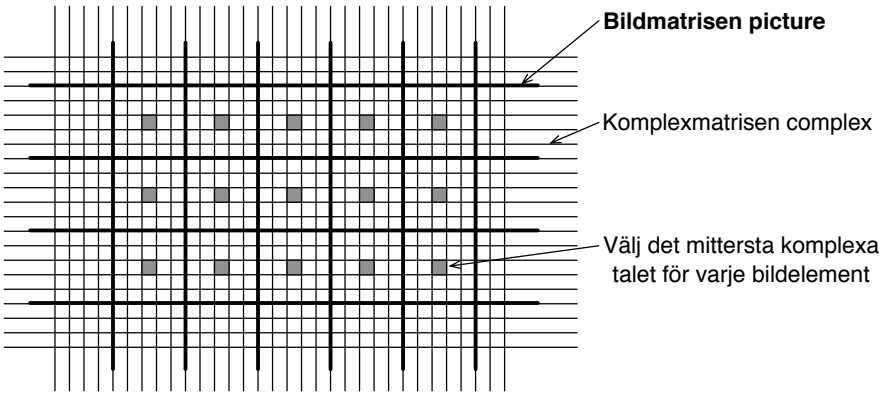
Som du ser är det är en hel del beräkningar att reda ut innan man kan skriva in och testa någon kod. Du sparar många felsökningstimmar vid datorn om du planerar noggrant med papper och penna.

Om bilden inte ritas upp rätt så beror det oftast på att man räknat fel i `mesh`. Skriv då om och testa igen — det är ingen idé att gå vidare förrän detta fungerar.

4 Zoom-funktionen

Du ska nu återgå till att skriva kod i `main`-metoden i `Mandelbrot`. Implementera `ZOOM` så att zoomning medför att bilden genereras om i det nya området. Detta ska dock bara göras om det redan finns en bild uppritad i fönstret. Om ritytan är tom, som den är när programmet nyss har startats eller efter en `RESET`-operation, ska inte `render` anropas. Testa!

[1] Man kan klara sig utan matrisen `complex` om man beräknar realdel och imaginärdel för rätt komplext tal när man behöver talet, dvs när man ska generera en bildpunkt. Beskrivningen i fortsättningen blir dock betydligt enklare om man använder sig av matrisen.



Figur 2: Medium upplösning — varje bildelement innehåller 5×5 pixlar.

5 Rita med olika upplösning

Användaren kan i en meny välja upplösning på den resulterande bilden i fem nivåer, från mycket låg upplösning till mycket hög. Vid mycket hög upplösning ska varje bildelement ha bredden 1 och höjden 1, vilket ger att matriserna `complex` och `picture` har samma storlek (det är denna upplösning vi använt i föregående avsnitt). Vid lägre upplösning ska bildelementen vara större, enligt följande tabell:

Resolution	Pixel width	Pixel height
Very high	1	1
High	3	3
Medium	5	5
Low	7	7
Very low	9	9

Bildmatrisen ska nu vara mindre än komplexmatrisen, på så sätt att man delar antalet rader och kolonner med de värden som finns i tabellen för en viss upplösning. Detta ger en mindre bildmatris och snabbare beräkningar. För varje bildelement ska man nu välja det mittersta komplexa talet — se figur 2, där upplösningen är Medium.

Ändra renderingsmetoden så att den tar hänsyn till den upplösning som användaren har valt. Provkör programmet och kontrollera att det fungerar — cirkelns periferi ska bli taggig i varierande grad beroende på upplösningen.

Du måste noggrant testa så att programmet fungerar med alla upplösningar. Felaktiga formler leder här ofta till att man råkar indexera utanför den ena eller den andra matrisens gränser, vilket naturligtvis inte är acceptabelt.

6 Rita Mandelbrotmängden

Nu ska du (äntligen) rita en bild av Mandelbrotmängden i stället för en cirkel. För vart och ett av c-värdena i bildelementens mittpunkter ska du beräkna en följd av komplexa tal med Mandelbrotformeln och med ledning av den sats som presenterades bestämma om punkten tillhör Mandelbrotmängden M eller inte. Använd förslagsvis högst 200 iterationer. Om punkten tillhör M sätter du bildmatrisens motsvarande element till `Color.BLACK`, annars till `Color.WHITE`. Gör inte fler iterationer än nödvändigt!

7 Avslutande förbättringar

Du ska förbättra din `Generator`-klass så att den uppfyller följande krav:

- Rita med färg om användaren har begärt det. I stället för att använda vit färg på alla divergerande punkter, låt dem få en annan färg som svarar mot ”hur snabbt” de divergerar.

  Av effektivitetsskäl bör man inte skapa nya färgobjekt varje gång man ska sätta en färg på en punkt. Det kan ju inte bli fråga om att använda fler än högst så många färger som man har iterationer, så dessa färger bör man skapa *en* gång och lägga i en vektor (lämpligen i konstruktorn i `Generator`). Jämför med vektorn `grayLevels` i deluppgift 5 i datorlaboration 8.

- Valfritt: utnyttja Extra-rutan i användargränssnittet för att mata in data till programmet, till exempel antalet iterationer som ska göras i Mandelbrotföljden (när man har zoomat in på fina detaljer i bilden är det bra att använda många iterationer), eller för att tala om vilken färgskala som ska användas, . . .

Rapport

I användargränssnittet finns ett kommando för att spara bilder som genererats. Skapa ett par bilder och inkludera dem i rapporten. Redogör i bildtexten för vilka inställningar (koordinataxlarnas intervallgränser, upplösning, osv) som har använts för bilden.

Användargränssnittet sparar bilder i JPEG-format. pdfLaTeX kan importera JPEG-bilder; om du använder ”vanlig” LaTeX måste bilderna vara i EPS-format. Använd i så fall ImageMagick-programmet `convert` för att konvertera:

```
convert picture1.jpg picture1.eps
```