Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 10 – ”Factories”

Designmönster med C++

Syfte:	Diskussion kring olika typer av Factory-objekt och introduktion av designmönstren Singleton , Object Pool Pattern och Factory Method.
Att läsa:	Design Patterns Explained kap 20-24, sid 347 – 396.

Kapitel 20 – ’Lessons from Design Patterns: Factories”

Ett Factory-objekt är ett objekt vars uppgift är att skapa andra objekt.

Den kategori av designmönster som GoF kallar ’creational’ innehåller designmönster vars gemensamma egenskap är att de erbjuder olika lösningar på problemet att skapa rätt typ av objekt vid rätt tillfälle.
Av dessa har du redan använt DP Abstract Factory.

De övriga är

- DP Builder
- DP Factory Method
- DP Prototype
- DP Singleton

Motivet för att använda speciella factory-object är att koden snabbt blir komplex om man låter samma kod som ska använda objekten också ansvara för att skapa dem. Det strider mot principen att alltid eftersträva ’strong cohesion’, dvs att varje kodavsnitt ska ha ett klart avgränsat ansvarsområde och inte omfatta flera orelaterade uppgifter.
Genom att koppla isär (decouple) ’skapare’ och ’användare’ blir också designprocessen enklare eftersom man primärt kan koncentrera sig på användningen av objekten och inte hur/var de skapas. Detta kan hanteras i ett senare skede. Bokens författare formulerar det i föjande regel:

”Concider what you need to have in your system before you concern yourself with how to create it…. That is, we define our factories after we decide what our objects are.”

Uppdelningen i ansvar mellan ‘skapare’ och ‘användare’ formuleras i regeln

”Objects should either make other objects or use other objects, but never both.”

Effekterna på koden blir

- ‘increased cohesion’
- ‘looser coupling’
- ‘increased testability’

När ändringar måste göras handlar det ofta om andra objekt ska användas eller att vissa objekt ska användas på ett annat sätt, mer sällan båda alternativen. Factories bidrar då till att begränsa arbetet till antingen till ’skapare’ eller ’användare’. Factories kan inte eliminera arbete men kan bidra till att begränsa komplexiteten.

Factories kan, utöver att skapa objekt, även användas för att hantera (manage) objekt, t.ex. hur många som ska skapas och om objekt eventuellt kan delas (sharing) mellan flera användare. Vi kan alltså kapsla in regler för vilka objekt som ska användas i ett factory-objekt.

Kapitel 21 – "The Singleton Pattern and the Double-Checked Locking Pattern"

DP Singleton Pattern och Double-Checked Locking Pattern löser samma problem – att garantera att endast ett objekt av en viss klass blir instansierat. Singleton är det som presenteras av GoF medan Double-Checked Locking Pattern är en variant av Singleton som även kan användas när ett objekt kan skapas 'samtidigt' av flera parallella trådar.

DP Singleton

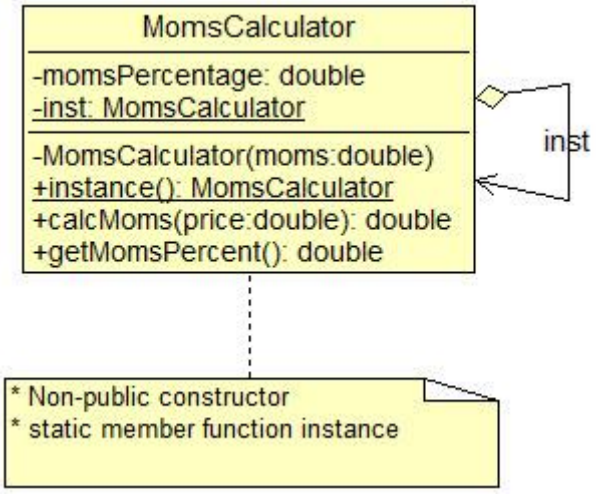
Syfte:
" *Ensure a class only has one instance, and provide a global point of access to it*." (GoF)

Ibland är det viktigt att det finns, och bara kan finnas, EN instans av en klass i en applikation. T.ex. en utskriftshanterare eller en sessionshanterare i kommunikation med en databas. Samtidigt ska den vara lättåtkomlig överallt i applikationen. Ett globalt objekt (i Java en instansvariabel som är public och static) är lättåtkomligt men saknar *garantier* för att vara den enda instansen.

Lösningen är att klassen själv håller kontroll på sin enda instans genom att ha

- en icke-public konstruktor
- en speciell accessmetod som
  - returnerar instansen om den redan finns
  - skapar den och returnerar den om den ännu inte finns
- access-metoden definieras som en statisk metod
  → accessbar utan instans.

Exempel (MomsCalculatorTest.cpp):
Beräkning av moms med ett MomsCalculator-objekt. Klassen är implementerad som en Singleton. Access till (den garanterat enda) instansen fås genom den statiska metoden `MomsCalculator::instance()`.




```
const double MOMSPERCENTAGE = 25.0;

class MomsCalculator {
public:
    // Static function ==> invocation without instance
    static MomsCalculator* instance() {
        if(inst==nullptr) {
            // Create it, if not already done.
            inst = new MomsCalculator(MOMSPERCENTAGE);
        }
        return inst;
    }

    double calcMoms(double price) {
        return momsPercentage * price / 100.0;
    }

    double getMomsPercent() {
        return momsPercentage;
    }


private:
    /* Construktor is non-public */
    MomsCalculator(double moms= MOMSPERCENTAGE)
    :momsPercentage(moms)
    { }

    // Accessed by static instance() ==> must be static
    static MomsCalculator *inst; // The instance
    double momsPercentage;

};

// Initialize the static pointer inst
MomsCalculator *  MomsCalculator::inst = nullptr;

class MomsCalcTest {
public:
```

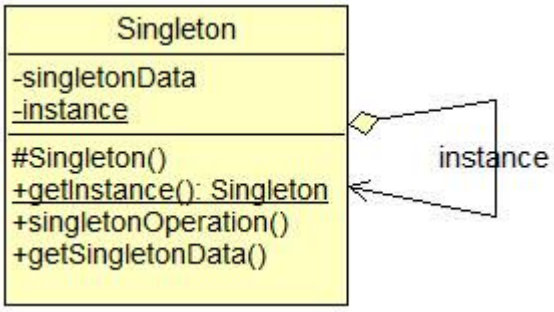
```
      void run() {
         double price;
         cout << "Price: ";
         cin >> price;
        while (price >0) {
           double moms = MomsCalculator::instance()-
>calcMoms(price);
           double percent = MomsCalculator::instance()-
>getMomsPercent();
           cout << "Moms: " << moms << " (" << percent << "%)" <<
endl;
           cout << "Price: ";
           cin >> price;
        }
     }
};

int main() {
    MomsCalcTest test;
    test.run();
    return 0;
}
```

Sammanfattning

DP **Singleton**

- Struktur



- Syfte
  - Att garantera att en klass endast har en instans och att ge en global accesspunkt till denna instans.

- Problem
  - Flera olika objekt behöver använda samma objekt och du vill vara söker på att du bara har en instans.

- Lösning
  - Låt Singleton ansvara för att endast en instans skapas.

- Deltagare
  - Klienter accessar en Singleton enbart genom den statiska metoden getInstance( )

- Konsekvenser
  - Kontrollerad access till den enda instansen.
  - Klienter behöver inte befatta sig med frågan om det finns någon instans av Singleton eller inte.
  - Tillåter ett variabelt antal instanser. Det är lätt att tillåta fler än en instans om man så vill..

- Använd Singleton när
  - det får finnas bara en instans av en klass och den ska vara accessbar för klienter via en känd access-punkt.
  - den enda instansen ska vara möjlig att sub-klassa och klienter ska kunna använda den sub-klassade instansen utan att modifiera sin kod

Om man vill deallokera en Singleton och göra det möjligt att skapa den på nytt vid behov kan tillföra en medlemsfunktion som återlämnar minnet och återställer instanspekaren till `nullptr`:

```
...
void destroySingleton() {
      if (inst) {
            delete inst;
            inst = nullptr;
      }
}
...
```

Singleton som klassmall

I C++ kan vi även utnyttja typparametrisering för att konstruera en Singleton:

```
template <typename T>
class Singleton
{
public:
      /* Return a reference to the instance of the singleton class.
         Create the instance if necessary */
      static T* instance(){
            if (inst == nullptr) inst = new T;
            return inst;
      };

private:
      // Non-public constructor.
      Singleton(){ }
```

```
      virtual ~Singleton(){ }
      Singleton(const Singleton& source) = delete; // No copy
constructor
      static T* inst; // Pointer to class instance
};

//Static class member initialisation.
template <typename T> T* Singleton<T>::inst = nullptr;
```

Se TemplateMomsCalcTest.cpp.

Singleton med lokal statisk variabel

En annan implementation av Singleton bygger på det faktum att en lokal statisk variabel skapas och initieras när programmet laddas och innan exekveringen startar. Detta leder oss till följande implementation av vår Singleton momskalkylator:

```
class MomsCalculator {
public:
    // Static function ==> invocation without instance
    static MomsCalculator* instance() {
           static MomsCalculator instObject{ MOMSPERCENTAGE };
        return &instObject;
    }

    double calcMoms(double price) {
        return momsPercentage * price / 100.0;
    }

    double getMomsPercent() {
        return momsPercentage;
    }


private:
    /* Construktor is non-public */
    MomsCalculator(double moms= MOMSPERCENTAGE)
    :momsPercentage(moms)
    { }

    // Accessed by static instance() ==> must be static
    static MomsCalculator *inst; // The instance
    double momsPercentage;

};
```

Vår Singleton kommer här att existera under hela programmets livslängd och kan följaktligen inte deallokeras

Se TemplateMomsCalcTestStatic.cpp.

”Trådsäker Singleton” Double-Checked Locking Pattern (DCLP)”

Ingen av de ovanstående varianterna av Singleton är trådsäker (”thread-safe”). För t.ex. originalversionen av en Singleton `()`, *kan det,* under vissa omständigheter (parallella anrop från flera trådar och en olycklig sekvens av ”thread-switch”), leda till att flera instanser av av Singleton-klassen skapas. Detta är framförallt ett problem om klassen i fråga har ’state’ dvs instans-variabler. Denna insikt ledde fram till utvecklandet av den ”trådsäkra” varianten av Singleton, Double-Checked Locking Pattern.

Algoritm:

```
...
if (instance == nullptr) { // Flera trådar kan passera här...
synkroniserad kod { // men endast en tråd i taget kan exekvera blocket
   if (instance == nullptr) // Kolla så ingen annan har gjort jobbet
       instance = new ClassInstance();// Nej, jag ska göra det
}
return instance;
```

Med `synkroniserad kod` menas att blocket skyddas av någon synkroniseringsmekanism som t.ex. mutex eller semafor som tvingar fram en serialiserad access till koden i blocket. En synkronisering utgör en flaskhals i exekveringen men den utförs i det här fallet bara första gången. Därefter är det bara den inledande if-satsen som testas, precis som i en ”vanlig” Singleton. Lösningen funkar fint i teorin och även i praktiken i C++ men efter något års användning i Java så uppdagades det under stort buller och bång att DCLP inte fungerade som det var tänkt. Anledningen visade sig prestandahöjande optimeringar i JVM (Java Virtual Machine). En ’work-around’ som baseras på Javas klass-laddare har sedan utvecklats.

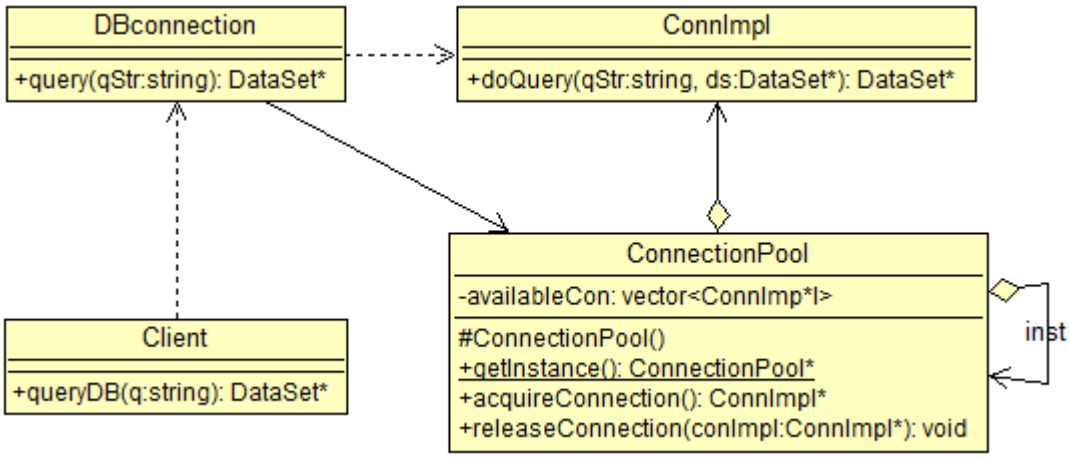
Kapitel 22 – ”The Object Pool Pattern (OPP)”

’Object Pool Pattern’ tillhör inte de grundläggande designmönster som presenterades av GoF men implementeras som en Singleton. OPP erbjuder en lösning på problemet att fördela en begränsad pool av återanvändbara resurser mellan klienter. Klienterna befrias från uppgifterna att själv allokera och administrera resurserna. Dessa uppgifter kapslas istället in i ’Reusable Pool’.

Boken presenterar ett exempel på hur det kan användas.

Här följer att annat exempel (se DBclient.cpp för en kodskiss).

En viss databas har en gräns för hur många klienter som samtidigt kan vara uppkopplade. En sådan uppkoppling representeras av ett ConnImpl-objekt. Eftersom uppkopplingen är ’state-less’ kan den återanvändas för flera frågor och av olika klienter. ConnectionPool skapar och administrerar ett visst antal ConnImpl-objekt.
När en klient ska ställa en databasfråga använder den ett DBconnection-objekt. DBconnection frågar ConnectionPool efter en ledig uppkoppling med `ConnectionPool->acquireConnection()` och när databasfrågan har returnerat sitt dataset återlämnas uppkopplingen till poolen med `ConnectionPool->releaseConnection(ConnImpl)`. För klienten blir alltså hanteringen av de begränsade resurserna i form av ConnImpl-objekt helt transparent[1]. En tillfällig brist på uppkopplingar märks endast som en fördröjning i anropet till `acquireConnection()` tills någon annan klient återlämnat sin uppkoppling eller ConnectionPool eventuellt lyckats allokera en ny.



[1] Jag blir inte klok på ordet ’transparent’. I datasammanhang står det för att någonting inte märks och att man inte behöver bekymra sig om det, ungefär det som är i en svart låda. Den egentliga betydelsen av transparent är ju den helt motsatta: det som är transparent är genomskinligt och erbjuder full insyn! Hmm...

Kapitel 23 – ”The Factory Method Pattern ”

Syfte:
” *Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses* .” (GoF)

Används ofta i samband med frameworks. Frameworks motsvarar en abstrakt nivå som inte kan bestämma exakt vilka konkreta objekt som ska skapas. Superklasser vet att objekt måste skapas men beslutet om exakt typ överlåts till subklasserna.

Factory Method kan användas för att knyta ihop parallella klasshierarkier, Klasserna i den ena hierarki kan använda Factory Methods för att instansiera objekt från den parallella hierarkin. Ett exempel på detta är DP Abstract Factory.

Exempel: MazeGame (exMazeGame)

I ett spel ska vi använda en labyrint (Maze) bestående av flera rum med en eller flera dörrar till andra rum.
Vi vill kunna skapa olika varianter av spelet…

MazeGame, den ’normala’ varianten:
– rum (Room) med väggar (Wall) och dörrar (Door)

Specialiserad variant:
BombedMazeGame:
– rum med bomber (RoomWithABomb) och bombade väggar (BombedWall)

En annan specialiserad variant:
EnchantedMazeGame
– ’förtrollade’ rum (EnchantedRoom) med förtrollade dörrar (DoorNeedingSpell)

… **men** behålla den grundläggande layouten för labyrinten.

Olika varianter av spelet använder olika ’byggstenar’ (deriverade från MazePart) för att bygga upp labyrinten

→ vi kan inte hårdkoda exakt typ för delarna men däremot bestämma ett *interface för att skapa delarna* med hjälp av *Factory Methods.*

```
// Factory methods
virtual Maze* makeMaze() {
      return (theMaze = new Maze);
}
virtual Room* makeRoom(int n) const {
      return new Room(n);
}
virtual Wall* makeWall() const {
      return new Wall;
}
virtual Door* makeDoor(Room* r1, Room* r2) const {
      return new Door(r1, r2);
}
```

MazeGame::createMaze( ) använder dessa metoder för att skapa en labyrint med en bestämd layout men utan att bestämma *exakta* typer för delarna:

```
// createMaze uses factory methods to create a maze.
Maze* MazeGame::createMaze() {

      Maze *aMaze = makeMaze();
      Room *r1 = makeRoom(1);
      Room *r2 = makeRoom(2);
      Room *r3 = makeRoom(3);
      Door *d1 = makeDoor(r1, r2);
      Door *d2 = makeDoor(r2, r3);
      d2->open();

      r1->setSide(North, makeWall());
      r1->setSide(East, d1);
      r1->setSide(South, makeWall());
      r1->setSide(West, makeWall());
      r2->setSide(North, makeWall());
      r2->setSide(East, makeWall());
      r2->setSide(South, d2);
      r2->setSide(West, d1);
      r3->setSide(North, d2);
      r3->setSide(East, makeWall());
      r3->setSide(South, makeWall());
      r3->setSide(West, makeWall());

      aMaze->addRoom(r1);
      aMaze->addRoom(r2);
      aMaze->addRoom(r3);

      return aMaze;
}
```

De specialiserade varianterna av spelet bestämmer själva exakt vilka objekt som ska skapas genom att omdefiniera vissa Factory methods, t.ex.

```
class BombedMazeGame : public MazeGame {
public:
      BombedMazeGame() { };
      Wall* makeWall()  const
      {
            return new BombedWall;
      }
      Room* makeRoom(int n)  const
      {
            return new RoomWithABomb(n);
      }
};
```

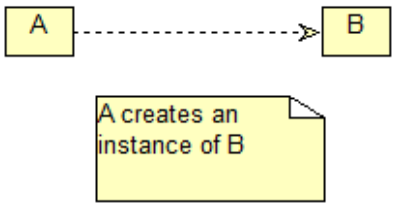
För att skapa ett ’vanligt’ MazeGame:

```
MazeGame *mazeGame = new MazeGame;
mazeGame->createMaze();
```

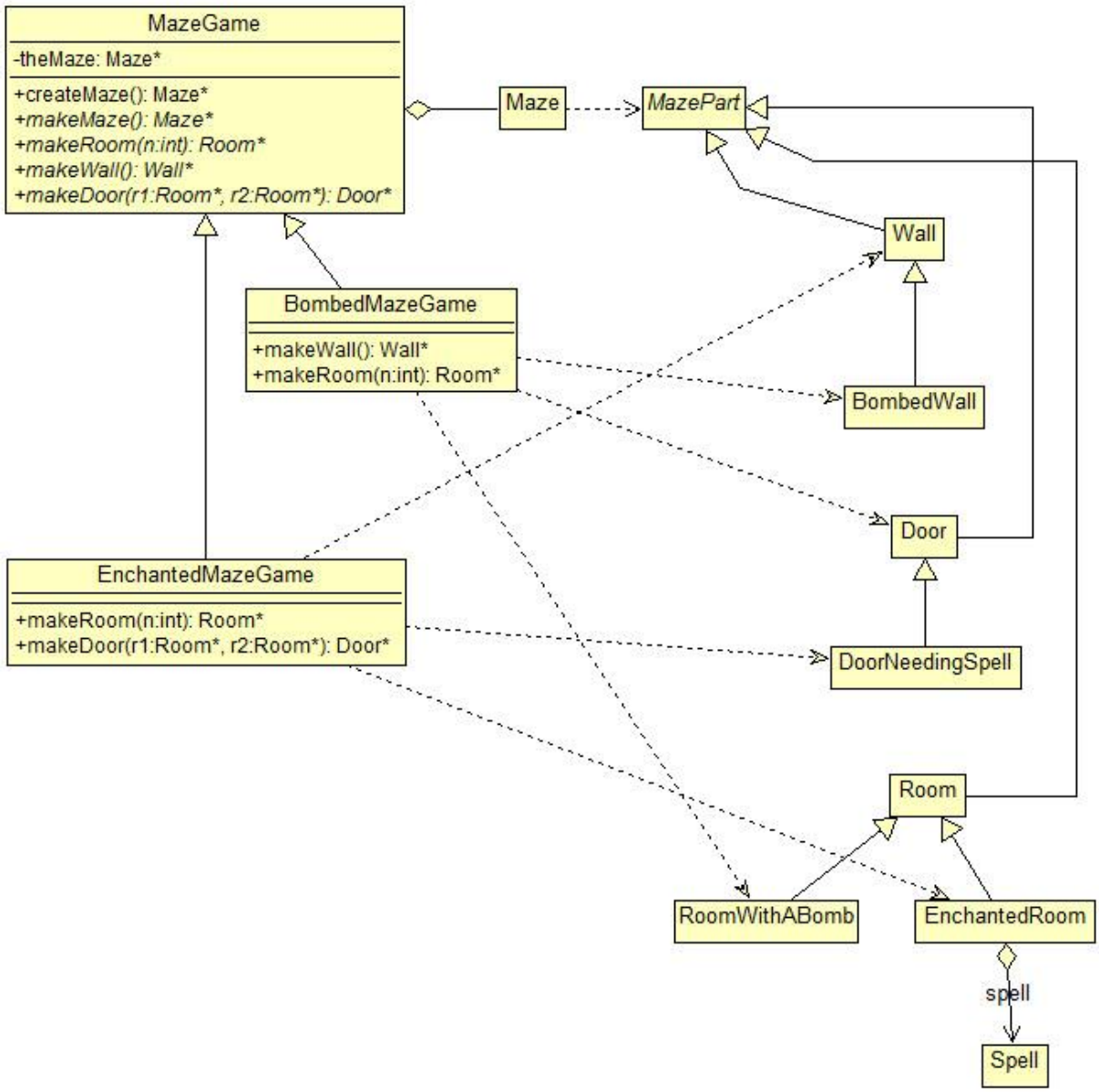
För att skapa ett MazeGame med väggar av typen `BombedWall` och rum av typen `RoomWithABomb`:

```
MazeGame *bombedMazeGame = new BombedMazeGame;
bombedMazeGame->createMaze();
```

I klassdiagrammet på följande sida används följande symbol för instansiering:



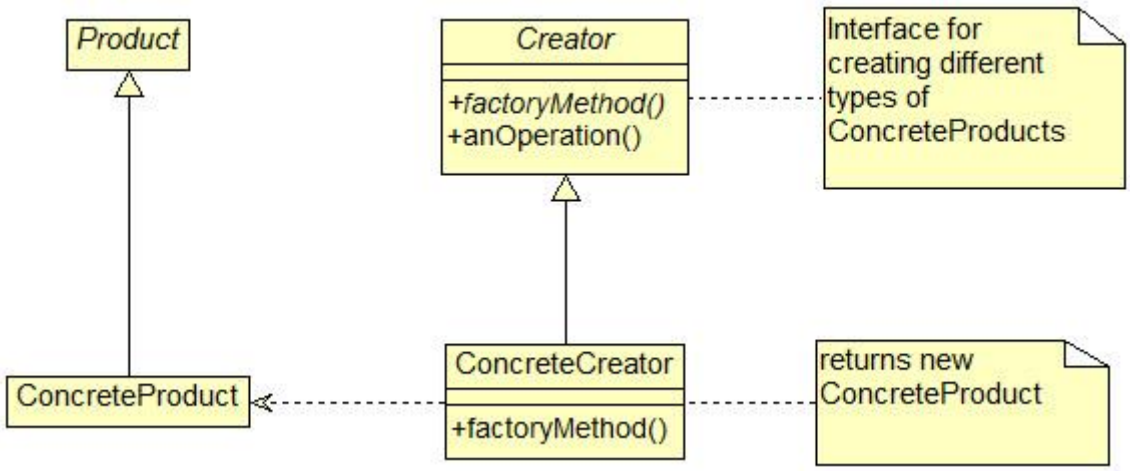
Här följer klassdiagrammet som illustrerar användningen av Factory methods i exemplet MazeGame.



Metoden createMaze implementeras i MazeGame och factory methods ger där labyrinten en bestämd struktur med ’standardelement’. De använda elementen (MazePart) i labyrinten kan varieras i subklasserna genom att dessa omdefinierar en eller flera factory methods. Om subklasserna ska tillåtas att ändra labyrintens struktur genom att omdefiniera metoden createMaze måste även den göras virtuell.

Sammanfattning Factory Method DP ('Virtual Constructor')

- Struktur



- Syfte
  - Att definiera ett interface för att skapa ett objekt men låta sub-klasser bestämma vilken klass som ska instansieras. Factory metod skjuter upp instansieringen och överlåter ansvaret till sub-klasser.

- Problem
  - En klass behöver instansiera en klass från en annan hierarki men vet inte exakt vilken.

- Lösning
  - En deriverad klass bestämmer vilken klass som ska instansieras och hur.

- Deltagare
  - Creator är interfacet som definierar Factory method. Product är interfacet för den typ av objekt som Factory Method skapar.

- Konsekvenser
  - Klienter måste sub-klassa (eller implementera) Creator för att skapa en viss ConcreteProduct

- Implementation
  - Använd en eller flera Factory Methods i abstrakt Creator-klass. Factory Method i en ConcreteCreator instansierar en ConcreteProduct. I C++ kan templates användas för att minska behovet av sub-klassning.
- Använd Factory Method när

- en klass inte kan förutse klassen för det objekt den ska skapa.
- en klass vill att dess sub-klasser ska specificera vilka objekt som ska skapas.

Förutom Singleton, Abstract Factory och Factory Method är Prototype och Builder ’Creational’ designmönster från GoF.