Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 5 – DP Abstract Factory

Designmönster med C++

Syfte: Härledning av DP Abstract Factory

Att läsa: Design Patterns Explained kapitel 11 sid 193 – 213.

Kapitel 11 'The Abstract Factory Pattern'

DP Abstract Factory

Syfte:
" *Provide an interface for creating families of related or dependent objects without specifying their concrete classes*." (GoF)

Innebörden är att en familj av abstrakta klasser kan implementeras av olika konkreta familjer av klasser. T.ex. har olika GUIer (GNOME, Windows, KDE osv) olika implementationer av Widgets: fönster, knappar, scroll-bars mm. Till varje GUI hör dock en viss uppsättning *konkreta klasser* som *implementerar* olika Widgets på ett specifikt sätt.

Hur ska vi designa en applikation som fungerar i flera olika fönstersystem och varje GUI har sitt eget "kit" av klasser?

En lösning är att definiera en *abstrakt* WidgetFactory som ger ett *interface* för att skapa varje typ av Widget.

- Varje GUI-typ har sin egen *konkreta* Factory som *implementerar* interfacet från abstrakta WidgetFactory.
- Varje typ av Widget motsvaras av en abstrakt klass.
- Varje GUI har sina egna konkreta klasser som implementerar olika Widgets.

Bokens motiverande exempel.

Design av ett system för att kunna presentera CAD-ritningar på skärmen och även skriva ut dem.

- Upplösningen (High Resolution/Low Resolution) på skärm och för skrivare beror på aktuell hårdvara och måste anpassas dynamiskt.
- Eventuellt ska en HR-driver kunna mixas med en LR-driver.

	Standard-PC	Monster-PC
Skärm	LRDD (Low resolution display driver)	HRDD (High resolution display driver)
Skrivare	LRPD (Low resolution print driver)	HRPD (High resolution print driver)

Tre ansatser testas.
Ansats 1: Switch-satser

```
class ApControl { // Controls drivers
   void doDraw() {
         switch (RESOLUTION) {
         case LOW:  // use LRDD
               case HIGH; // use HRDD
         }
   }

   void doPrint() {
         switch (RESOLUTION) {
         case LOW:  // use LRPD
               case HIGH; // use HRPD
         }
   }
   ...
};
```

Fungerar men lider av:

- *Tight Coupling* - att införa en ny upplösning, MIDDLE, skulle leda till ändringar i koden på två ställen som för övrigt är oberoende av varandra.
- *Weak Cohesion* – både doPrint och doDraw måste utföra två orelaterade uppgifter:
  - skapa en ritning och
  - ta ställning till lämplig driver
  Dessa har egentligen inget med varandra att göra.

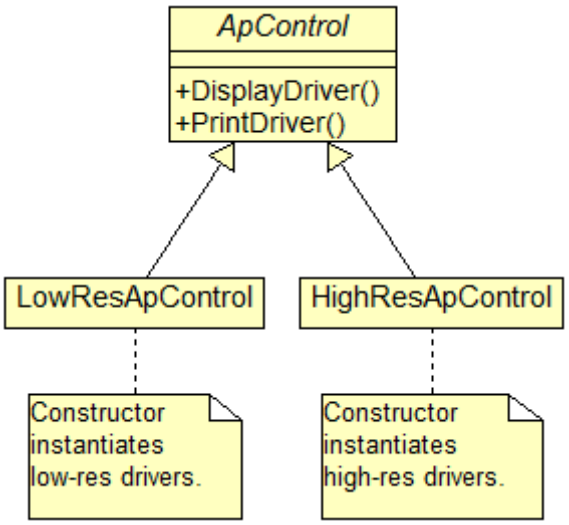
I förlängningen kommer detta att försvåra underhåll.

Iakttagelse: switch-satser indikerar ofta

- Behov av polymorfism → försök hitta en abstraktion.
- Felplacerat ansvar → överför ansvar till annat objekt.

Ansats 2: Arv

Derivera olika kontrollklasser från en abstrakt basklass.

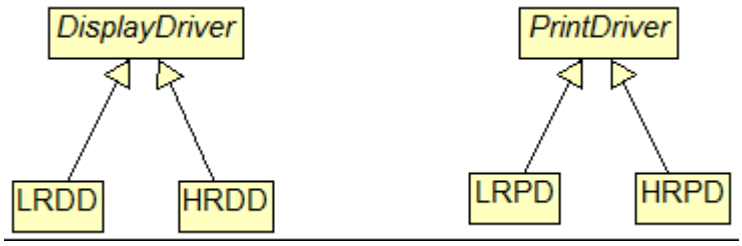


Fungerar i detta enkla fall men

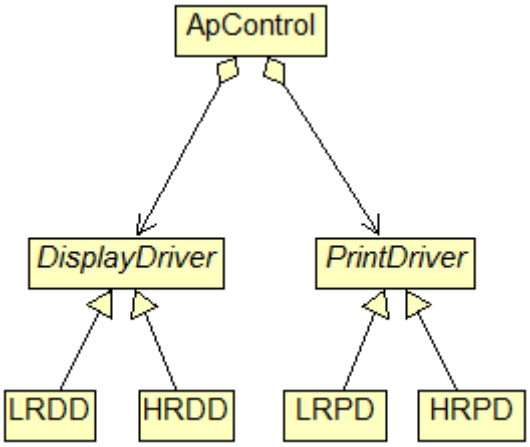
- Kan leda till kombinatorisk explosion, t.ex. ger kombinationer av HR- och LR-drivers många nya klasser
- Har oklar semantik – koden blir svår att förstå
- Bryter en grundläggande regel : "Välj komposition före arv”

Ansats 3: Ersätt switches med abstraktioner

LRDD och HRDD är båda drivrutiner för en display och LRPD och HRPD är båda drivrutiner för skrivare. Vi inför abstraktionerna DisplayDriver och PrintDriver → abstrakta klasser.



ApControl kan nu använda sig av DisplayDriver-objekt och PrintDriver-objekt:



ApControl kan nu använda sig av DisplayDriver- och PrintDriver-objekt utan att veta exakt vilka eller vilka upplösningar som används.

Obesvarad fråga i sammanhanget:
- ***Vem skapar*** de driver-objekt som ApControl ska utnyttja?

Om ApControl själv ska skapa objekten blir underhåll/utvidgning svårt eftersom ApControl alltid måste ändras när nya versioner av drivers tillkommer.

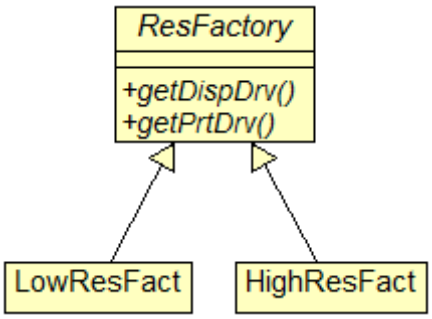
Ett bättre sätt är att överlåta ansvaret till ett speciellt ”Factory’-objekt”!

Vi överlåter till ett ResFactory-objekt att skapa de konkreta driver-objekten:
- ApControl behöver inte veta exakt typ på driver-objekten utan använder interfacet i ResFactory för att få tillgång till drivrutinerna.

- Ansvaret för att bestämma exakt vilka drivrutiner som ska användas kan flyttas från ApControl

  → Uppdelning av ansvar
  - Ett ResFactory-objekt instansierar de drivrutinerna som ska användas
  - ApControl använder dem

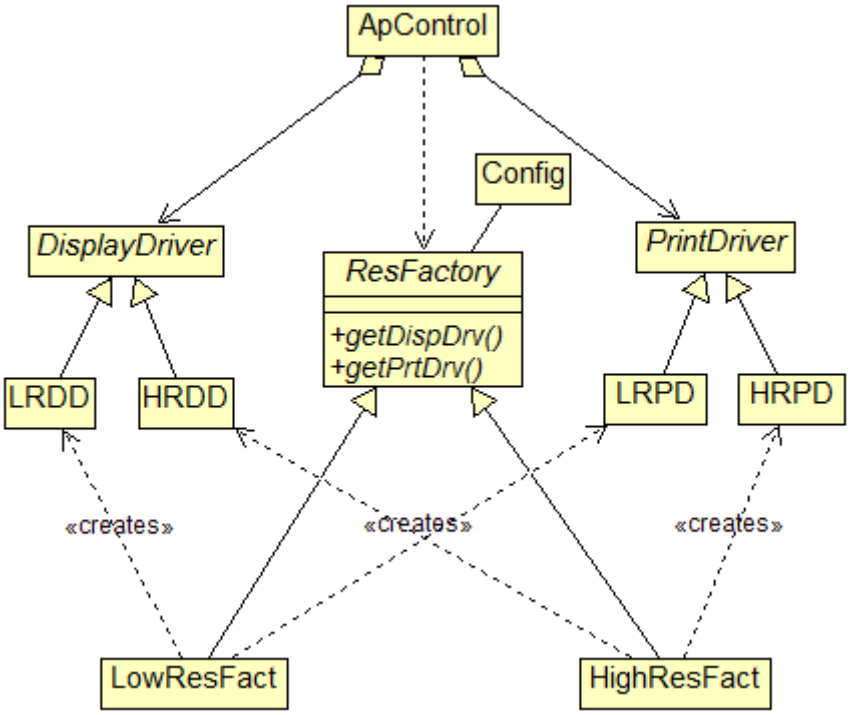
ResFactory är lämpligen en abstrakt klass med deriverade konkreta Factory-klasser:



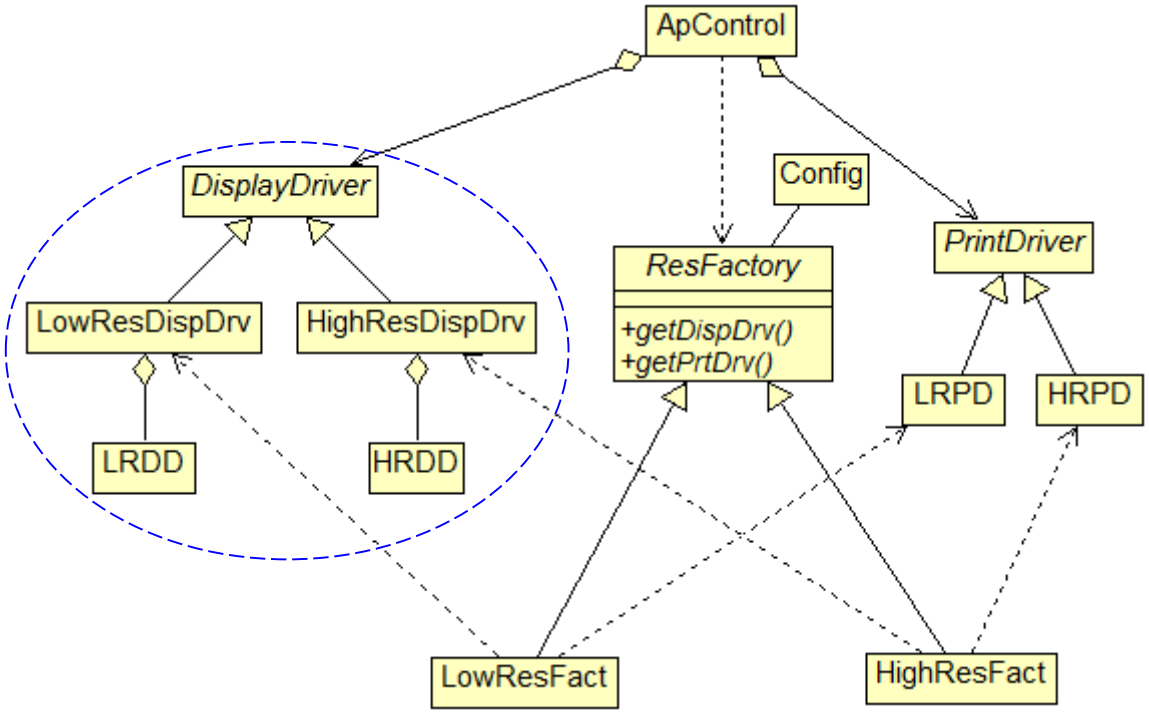
LowResFact/HighResFact implementerar interfacet i ResFactory och instansierar de drivrutiner som ska användas.

Valet av ’familj’ av drivrutiner görs av någon form av konfigurations-objekt baserad på en konfigurationsfil, miljövariabler eller systeminformation under körning. Konfigurationsobjektet instansierar rätt konkret ResFactory-objekt som sedan används av ApControl. *Det innebär att ApControl inte behöver veta vilken konkret ResFactory som används utan utnyttjar endast interfacet i ResFactory.*

Lösningen kan beskrivas i klassdiagrammet



Ett eventuellt problem kan vara att t.ex. LRDD och HRDD inte är deriverade från samma basklass och alltså inte implementerar samma interface. Detta kan vi lösa genom att införa ett mellanskikt med en objekt-adapter:

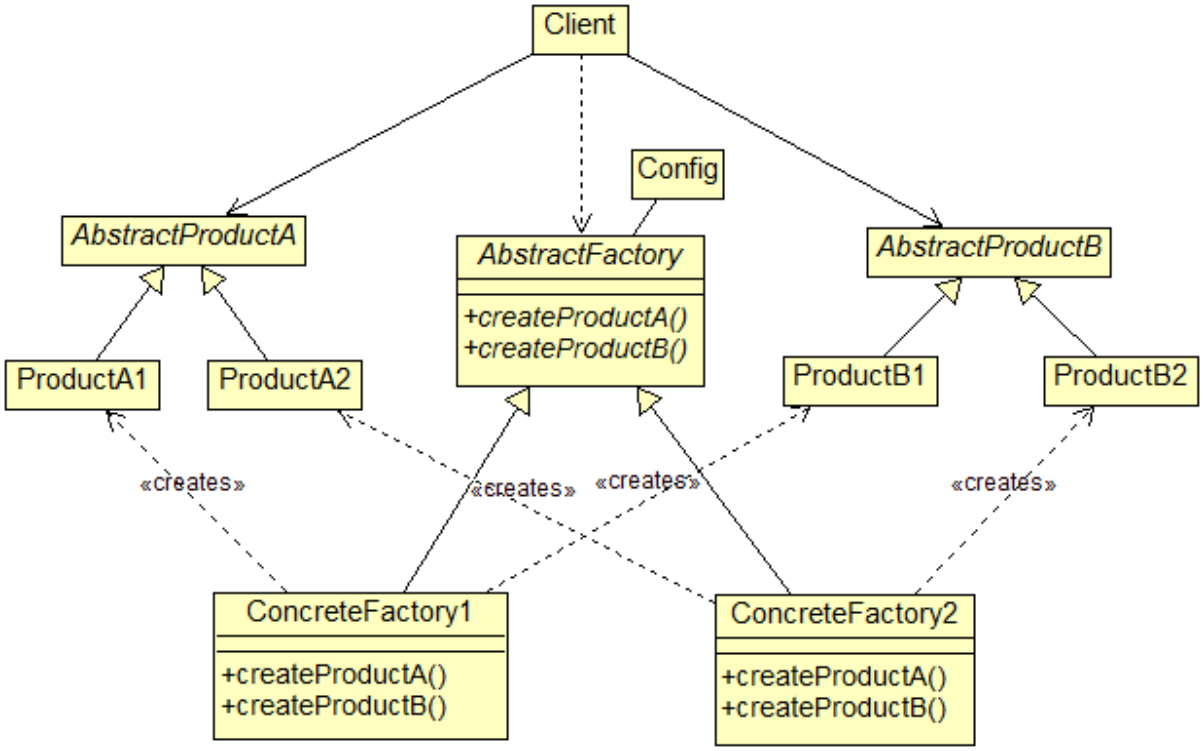


Lösningen är ett exempel på DP Abstract Factory och utnyttjar principerna

- "Find what varies and encapsulate it."
  - Valet av driver varierar → inkapslat i ResFactory
- "Favour aggregation over inheritance."
  - De olika variationerna återspeglas i de olika ResFactory-objekt som ApControl kan använda sig av. Detta i kontrast till att ha olika ApControl-klasser.
- "Design to interfaces, not to implementations."
  - ApControl vet hur man begär drivers från ResFactory men inte hur ResFactory genomför detta.

Sammanfattning DP Abstract Factory (’Kit’)

- Struktur



- Syfte
  - Erbjuda ett interface för att skapa familjer av besläktade eller beroende objekt utan att specificera deras konkreta klasser.

- Problem
  - Familjer av besläktade objekt ska instansieras.

- Lösning
  - Abstract Factory koordinerar instansiering av objekt. Erbjuder ett sätt att flytta regler för hur instansiering ska utföras från klienterna som ska använda objekten.

- Deltagare
  - Abstract Factory definierar hur varje medlem i en familj av klasser skapas. Vanligen skapas en familj genom att ha sin egen ConcreteFactory.

- Konsekvenser

  - Isolerar klienter från konkreta klasser. Klienter använder instanserna genom deras abstrakta interface.
  - Konkreta produktnamn isoleras till implementationen av en ConcreteFactory, namnen används inte i klientkoden.
  - Underlättar utbyte av en hel produktfamilj.
  - Upprätthåller konsistens mellan de använda objekten genom att de endast skapas familjevis.
  - Svårt att tillföra nya konkreta produkter i efterhand, interfacet till AbstractFactory och dess underklasser måste i så fall ändras.

- Implementation
  - Definiera en abstrakt klass som specificerar metoder för att skapa de olika objekten.
  - En konkret klass definieras för varje familj av konkreta produkter som ska skapas.
  - Eftersom en applikation endast behöver ett Factory-object för varje familj så implementeras Abstract Factory ofta som en Singleton.

- Använd Abstract Factory när
  - en familj av objekt ska användas tillsammans och du vill framtvinga detta.
  - du vill erbjuda ett klassbibliotek med produkter och du endast vill visa interfacen men inte implementationerna.