Patrons de disseny GoF

Juan Manuel Gimeno Illa

jmgimeno@diei.udl.cat

Curs 2008-2009

Patrons GoF

- El llibre *Design Patterns* de la *"banda dels quatre"* (Gamma, Helm, Johnson, Vlissides) publicat al 1995 va marcar una fita important al món del disseny orientat a objectes
- Els patrons de disseny descriuen solucions simples i elegants a problemes específics del DOO
- Representen solucions que han estat desenvolupades i que han evolucionat al llarg del temps
  - no representen solucions inicials i intuïtives
  - reflexen tot el redisseny que els desenvolupadors han fet mentre buscaven
    - ★ més reutilització
    - ★ més flexibilitat
- *"Cada patró descriu un problema que succeeix repetidament al nostre entorn així com la solució a aquest problema de manera que es pugui aplicar indefinidament sense fer el mateix dues vegades" (Alexander)*

El catàleg GoF

- El llibre descriu un total de 23 patrons amb diferents propòsits i àmbits
- El **propòsit** reflexa què fa el patró. N'hi ha 3 propòsits diferents:

  de creació tenen a veure amb el procés de creació d'objectes

  estructural tracten de la composició de classes i objectes

  comportament caracteritzen el mode en què les classes i objectes interactúen i es reparteixen les responsabilitats
- L'**àmbit** especifica si el patró s'aplica principalment a classes o objectes.

  classe relacions entre classes i les seves subclasses (herència → polimorfisme)

  objecte relacions entre objectes, que poden canviar-se en temps d'execució (associació → delegació)

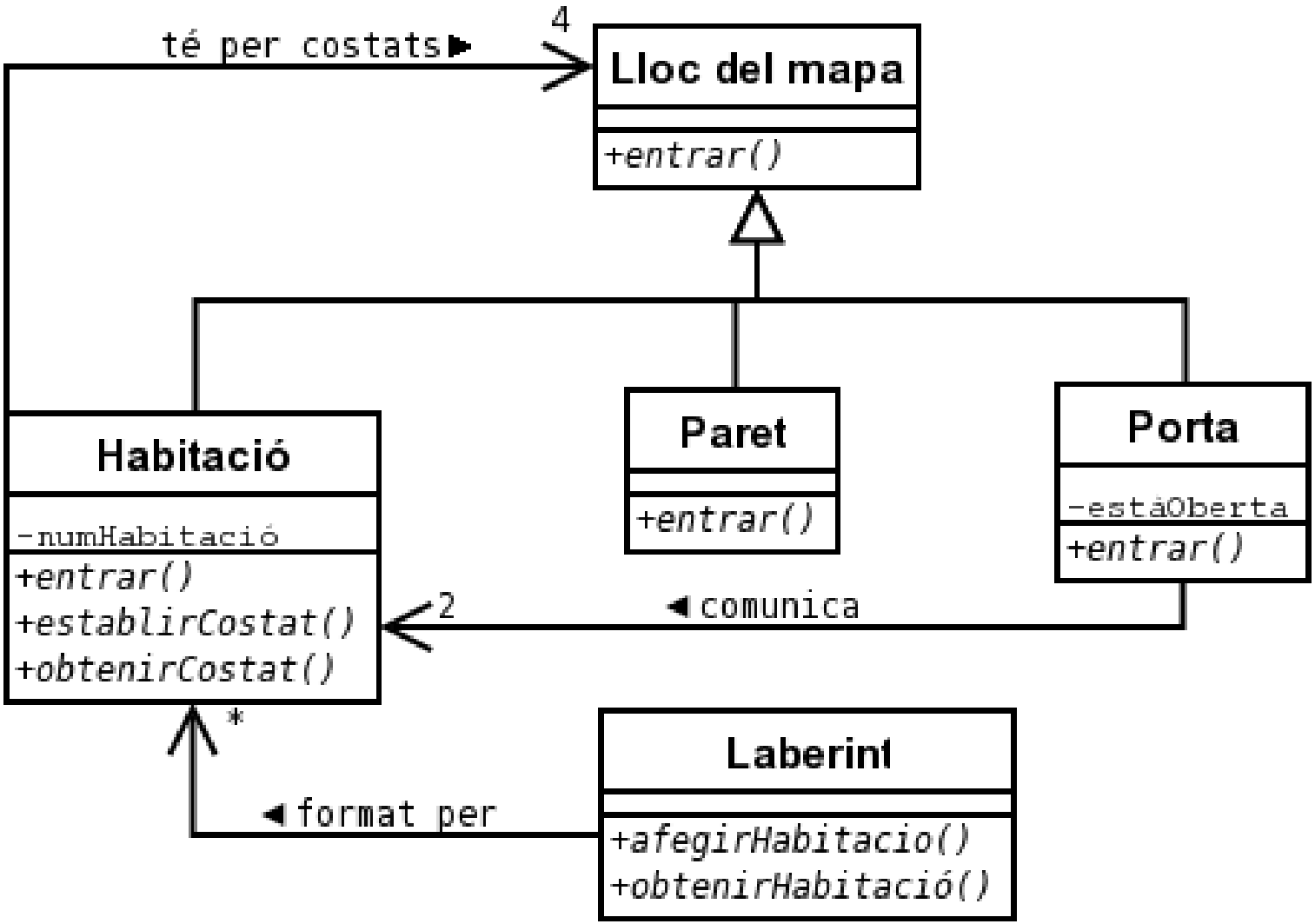
El catàleg GoF

	creació	**estructural**	**comportament**
classe	*factory method*	*adapter C*	interpreter *template method*
objecte	*abstract factory* *builder* *prototype* *singleton*	*adapter O* bridge *composite* decorator façade flyweight *proxy*	chain of responsibility command *iterator* mediator memento *observer* state *strategy* *visitor*

El joc del laberint

- El *joc del laberint* és l'exemple que segueix el llibre *"Patrones de diseño"* per exemplificar els diferents patrons de creació.
- Consisteix en la infraestructura per a un hipotètic joc d'aventures.
- El laberint consta d'una sèrie d'habitacions separades per parets o comunicades amb portes.
- Primerament veurem les classes que s'utilitzen per a representar els diferents elements d'un laberint així com el codi per a crear-ne un.
- (Al llibre la implementació està en `C++` però presentarem una versió `Java` d'aquest mateix codi)

Elements d'un laberint



Una mica de codi

```
enum Direcccio {NORD, SUD, EST, OEST}
abstract class LlocDelMapa {
    abstract void entrar();
}
class Habitacio extends LlocDelMapa {
   private EnumMap<Direccio,LlocDelMapa> costats;
   private int num_habitacio;
   public Habitacio(int numHabitacio) {...}
   public LlocDelMapa obtenirCostat(Direccio d) {...}
   public void establirCostat(Direccio dir,
                              LlocDelMapa lloc) {...}
   public void entrar() {...}
}
```

Més codi

```
class Paret extends LlocDelMapa {
  public Paret() {...}
  public void entrar() {...}
}
class Porta extends LlocDelMapa {
  private Habitacio habitacio1;
  private Habitacio habitacio2;
  private bool esta_oberta;
  public Porta(Habitacio h1, Habitacio h2) {...}
  public Porta(){this(null,null);}
  public void entrar() {...}
  public Habitacio altreCostatDe(Habitacio h) {...}
}
```

I una mica més

```
class Laberint {
  private .....;
  public Laberint() {...}
  public void afegirHabitacio(Habitacio h) {...}
  public Habitacio obtenirHabitacio(int num) {...}
}
```

- Aquestes classes són la base per a crear un laberint
- Per fer-ho, definirem una classe anomenada **JocDelLaberint** que crearà els seus components i els connectarà entre si.

Creant un laberint

```
class JocDelLaberint {
  public Laberint crearLaberint() {
    Laberint unLaberint = new Laberint();
    Habitacio h1 = new Habitacio(1);
    Habitacio h2 = new Habitacio(2);
    Porta laPorta = new Porta(h1,h2);
    unLaberint.afegirHabitacio(h1);
    unLaberint.afegirHabitacio(h2);
    h1.establirCostat(NORD, new Paret());
    h1.establirCostat(EST, laPorta);
    h1.establirCostat(SUD, new Paret());
    h1.establirCostat(OEST, new Paret());
    h2.establirCostat(NORD, new Paret());
    h2.establirCostat(EST, new Paret());
    h2.establirCostat(SUD, new Paret());
    h2.establirCostat(OEST, laPorta);
    return unLaberint;
  }
}
```

Comentaris de crearLaberint

- El problema d'aquesta funció és la seva **inflexibilitat**
- Com cridem directament als constructors dels objectes que formen part del laberint, si definim nous tipus de parets, habitacions, portes, etc. el codi que tenim no el podem aprofitar (ja que l'operador **new** no es pot comportar de forma polimòrfica)
- Els **patrons de creació** ens mostraran con fer un codi més **flexible** i que no invoqui directament els constructors.

Definicions

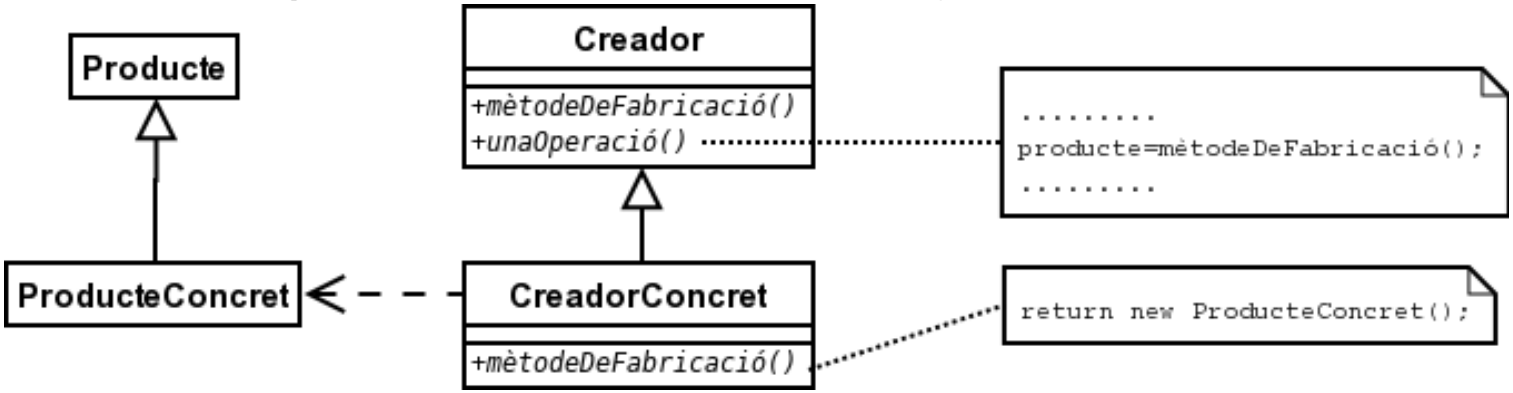
- Els patrons de creació tenen que veure amb el procés de **crear objectes**
- Es poden agrupar en **dues famílies** segons el seu **àmbit**

  Classe s'ocupen de les relacions entre les classes i les seves subclasses (o entre interfícies i les classes que les implementen). Aquestes relacions s'estableixen mitjançant **herència** (en Java via *extends* o *implements*) i, per tant, són **estàtiques**

  Objecte s'ocupen de les associacions entre objectes, que poden **canviar** en temps d'execució i, per tant, són **dinàmiques**

Factory Method

Defineix una **interfície** per a crear un objecte però deixa que siguin les **subclasses** les qui decideixin quina classe instanciar. Permet que una classe **delegui** la creació dels seus objectes a les seves subclasses



Producte Defineix la interfície dels objectes que crea el mètode de fabricació

ProducteConcret Implementa la interfície `Producte`

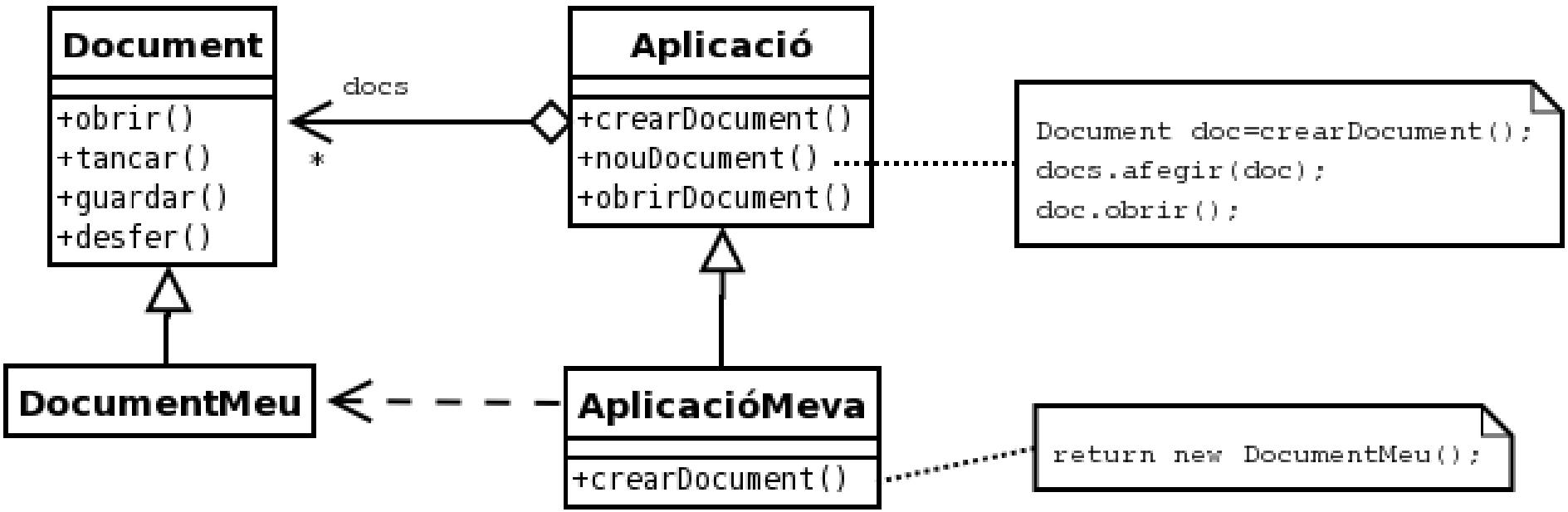
Creador Defineix el mètode de fabricació (que retorna un objecte `Producte`)

CreadorConcret Redefineix el mètode de fabricació per a retornar una instància de `ProducteConcret`

Observacions

- Els **mètodes de fabricació** eliminen la necessitat de lligar classes específiques de l'aplicació al nostre codi (`unaOperació` no sap quins objectes està creant)
- El codi només tracta amb la interfície `Producte`; a més pot funcionar amb qualsevol classe `ProducteConcret`
- Els clients han d'heretar de `Creador` només per a crear objectes de la classe `ProducteConcret` i això introdueix acoblament
- Proporciona ganxos per a les subclasses: crear objectes dintre d'una classe amb un mètode de fabricació dóna ganxos que les subclasses poden fer servir per a extendre l'objecte.

Exemple: framework d'aplicacions



Dins del laberint

Per tal de flexibilitzar la funció `crearLaberint`, primer introduirem **mètodes de fabricació** a la classe `JocDelLaberint`

```
class JocDelLaberint {
    public Laberint fabricarLaberint() {
      return new Laberint();}
    public Habitacio fabricarHabitacio(int n) {
      return new Habitacio(n);}
    public Paret fabricarParet(){
      return new Paret();}
    public Porta fabricarPorta(Habitacio h1,
                               Habitacio h2) {
      return new Porta(h1,h2);}
    ....
```

Redefinint crearLaberint

```
// continuacio de la classe JocDelLaberint
....
public Laberint crearLaberint() {
  Laberint unLaberint = fabricarLaberint();
  Habitacio h1 = fabricarHabitacio(1);
  Habitacio h2 = fabricarHabitacio(2);
  Porta laPorta = fabricarPorta(h1,h2);
  unLaberint.afegirHabitacio(h1);
  unLaberint.afegirHabitacio(h2);
  h1.establirCostat(Nord, fabricarParet() );
  h1.establirCostat(Est, laPorta);
  h1.establirCostat(Sud, fabricarParet() );
  h1.establirCostat(Oest, fabricarParet() );
  h2.establirCostat(Nord, fabricarParet() );
  h2.establirCostat(Est, fabricarParet() );
  h2.establirCostat(Sud, fabricarParet() );
  h2.establirCostat(Oest, laPorta);
  return unLaberint;
}
}
```

Aplicant la flexibilitat

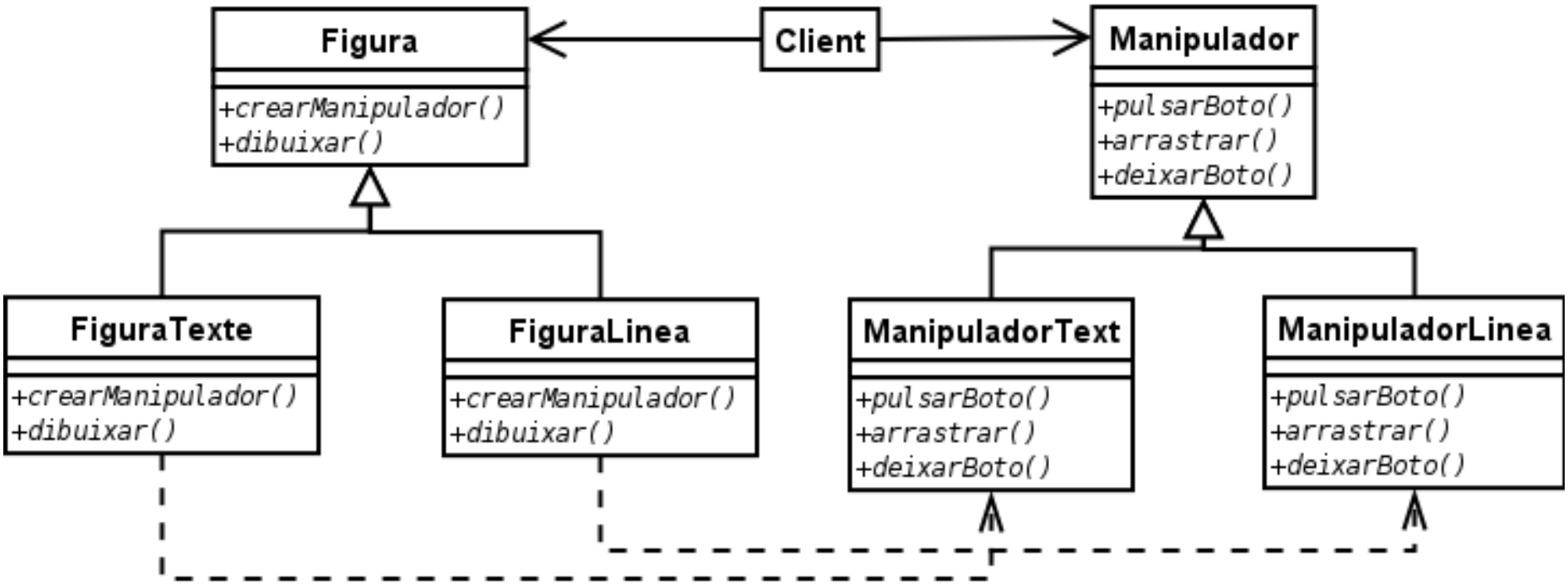
Jocs diferents poden heretar de `JocDelLaberint` per **especialitzar** parts del laberint

```
class JocDelLaberintEncantat
  extends JocDelLaberint {
  public Habitacio fabricarHabitacio(int n) {
    return new HabitacioEncantada(n, Encantar());}
  public Porta fabricarPorta(Habitacio h1,
                             Habitacio h2) {
    return new PortaEncantada(h1,h2);}
  protected Encanteri Encantar() {...}
}
```

Ara `crearLaberint` a `JocDelLaberintEncantat` crea el mateix laberint que abans (en quant a mapa) però encantat, no és encantador?

Jerarquies paral·leles

- Fins ara només hem vist casos en que el **mètode de fabricació** era utilitzat des de la classe `Creador`
- Les **jerarquies paral·leles** solen passar quan una classe **delega** alguna de les seves responsabilitats en una altra classe
- *E.g.*, jerarquia de figures i la seva manipulació.



Mètodes de fabricació i funcions virtuals

- Els MF són sempre mètodes virtuals i, per tant, **no** es poden cridar des del constructor del `Creador` (ja que el MF del `CreadorConcret` encara no estarà disponible).
- Es pot evitar això amb mètodes d'accés als productes que creen el producte quan és necessari. En comptes de crear el producte, el constructor l'inicialitza a `null`.
- Això es coneix com **inicialització mandrosa** (*lazy initialization*).

Inicialització mandrosa

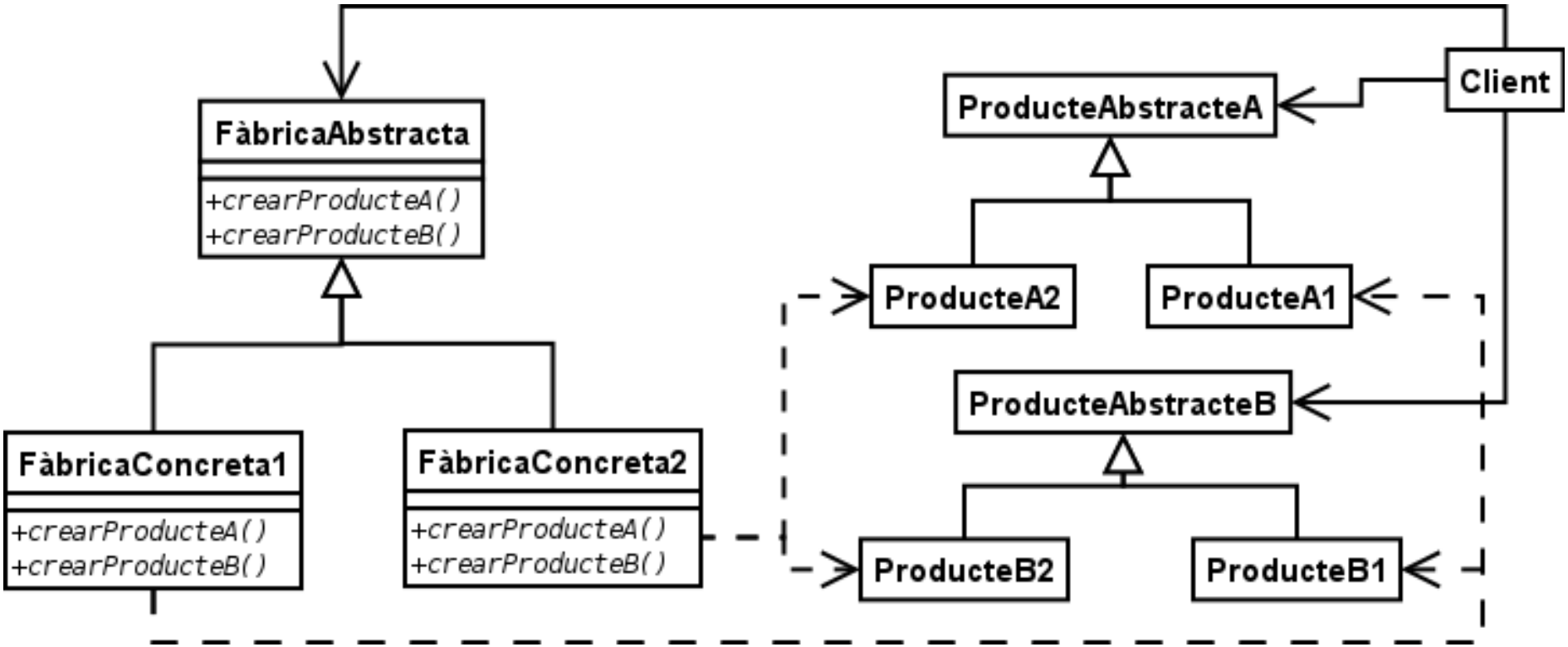
```
class Creador {
  private Producte producte;
  protected Producte crearProducte() {
    // crea i retorna el producte
    // pot redefinir-se a les subclasses
  }
  public final Producte obtenirProducte() {
    if (producte == null) {
      producte = crearProducte();
    }
    return producte;
  }
}
```

Mètodes de fabricació parametritzats

```
enum IdProducte {MEU, TEU}
class Creador {
  public Producte crearProducte(IdProducte id) {
    switch(id) {
      case MEU: return new ProducteMeu();
      case TEU: return new ProducteTeu();
    }
  }
}
class AltreCreador extends Creador {
  public Producte crearProducte(IdProducte id) {
    if (id == MEU) return new ProducteAltre();
    else return super.crearProducte(id);
  }
}
```

Abstract Factory

Proporciona una interficie per a crear famílies d'objectes relacionats o que depenen entre si, sense especificar les seves classes concretes



Participants

FàbricaAbstracta Defineix una interfície per les operacions que creen objectes producte abstractes

FàbricaConcreta Implementa les operacions per a crear objectes producte concrets

ProducteAbstracte Defineix una interfície per a un tipus d'objecte producte

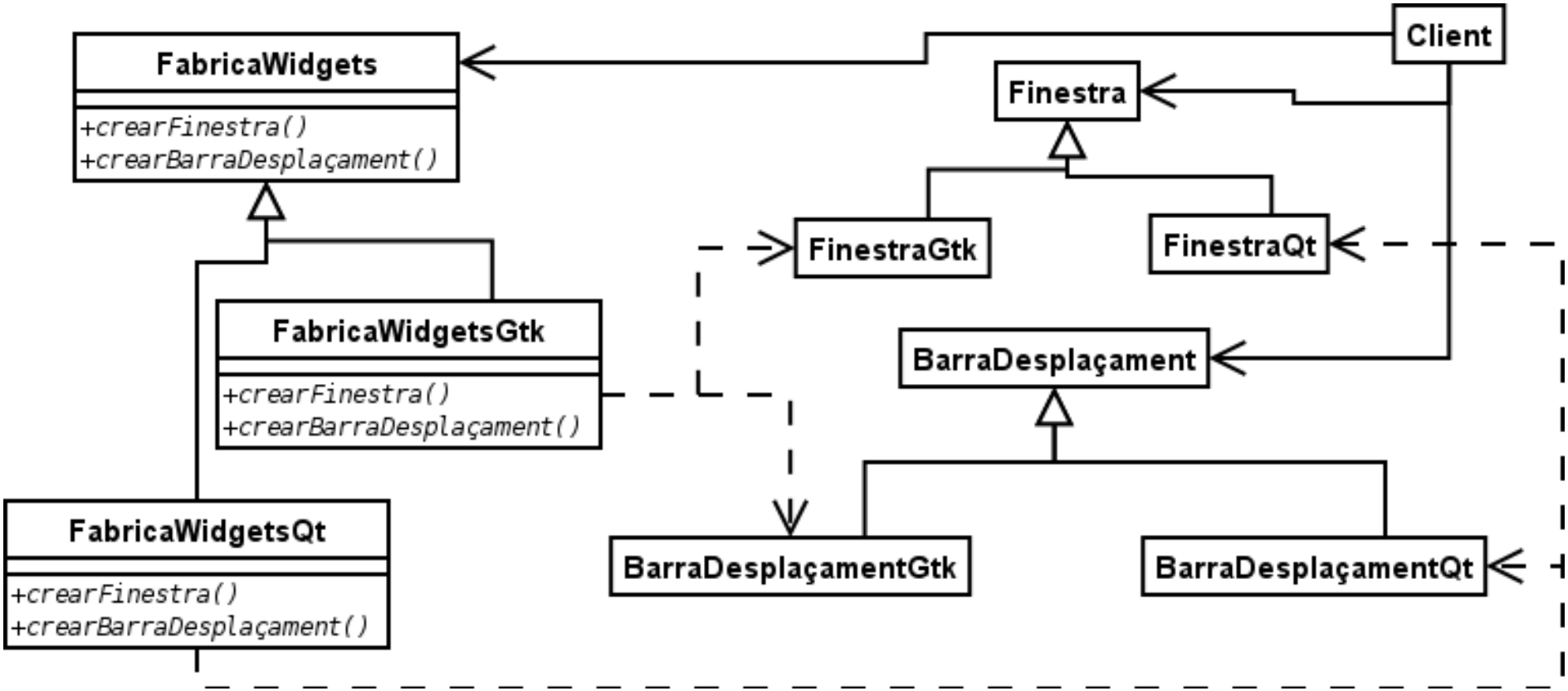
ProducteConcret Defineix un objecte producte per a que sigui creat per la fàbrica corresponent i implementa la interfície `ProducteAbstracte`

Client Només utilitza interfícies definides per les classes `FàbricaAbstracta` i `ProducteAbstracte`

Observacions

- Aïlla les classes concretes doncs la fàbrica abstracta encapsula la responsabilitat de crear els objectes producte
- Facilita l'intercanvi de famílies de productes doncs la fàbrica concreta apareix **una sola vegada** en tota l'aplicació i crea la família completa de productes. Si es canvia aquesta fàbrica concreta, **tota** la família de productes canvia de cop
- Promou la **consistència** entre productes ja que cada fabrica concreta els agrupa de forma coherent
- Un problema és que **ampliar la gamma de productes és difícil** doncs cal canviar la interfície de la fàbrica abstracta i ampliar les implementacions de cada fàbrica concreta

Exemple: Toolkit interfícies



Creant una fàbrica pels laberints

Definirem una classe anomenada `FabricaDeLaberints` que creï els diferents components.

```
class FabricaDeLaberints {
    public Laberint ferLaberint() {
      return new Laberint();}
    public Paret ferParet() {
      return new Paret();}
    public Habitacio ferHabitacio(int n) {
      return new Habitacio(n);}
    public Porta* ferPorta(Habitacio h1,
                           Habitacio h2) {
      return new Porta(h1,h2);}
}
```

Utilitzant la fàbrica a crearLaberint

Ara modificarem la funció `JocDelLaberint::crearLaberint` de manera que rebi com a paràmetre un objecte `FabricaDeLaberints` i l'utilitzi per a crear els laberints

```
class JocDelLaberint {
  public Laberint crearLaberint(FabricaDeLaberints fab) {

    Laberint unLaberint = fab.ferLaberint();
    Habitacio h1 = fab.ferHabitacio(1);
    Habitacio h2 = fab.ferHabitacio(2);
    Porta laPorta = fab.ferPorta(h1,h2);

    unLaberint.afegirHabitacio(h1);
    unLaberint.afegirHabitacio(h2);

    h1.establirCostat(Nord, fab.ferParet());
    ..........................................
    return unLaberint;
  }
}
```

Definint una nova fàbrica

Podem crear una `FabricaDeLaberintsEncantats` per tal de definir aquest nou tipus de laberints com a subclasse de `FabricaDeLaberints`

```
class FabricaDelLaberintsEncantats
  extends FabricaDeLaberints {

    public Habitacio ferHabitacio(int n) {
      return new HabitacioEncantada(n, Encantar());}

    public Porta ferPorta(Habitacio h1,
                          Habitacio h2) {
      return new PortaEncantada(h1,h2);}

    protected Encanteri Encantar() {...}
}
```

Mostrant la nova flexibilitat

Depenent de l'objecte que passem a `crearLaberint` crearem un tipus de laberint o un altre

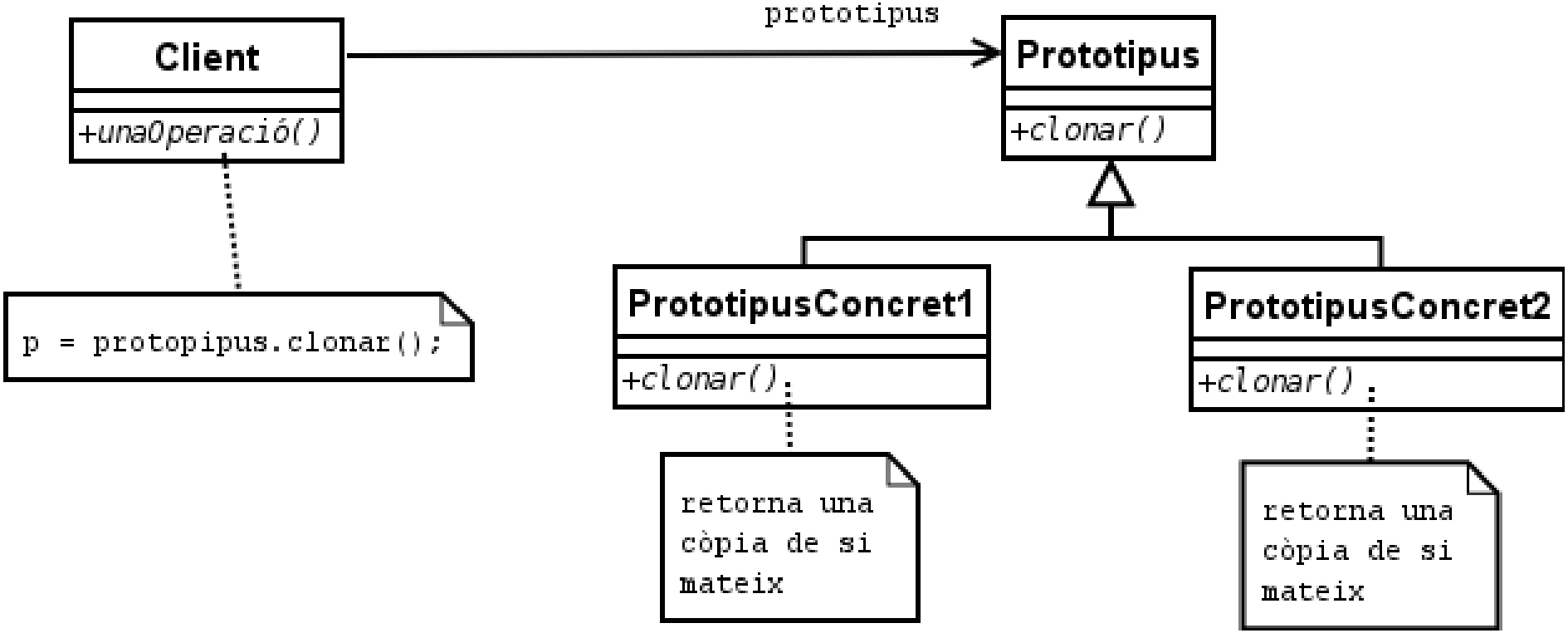
```
// Defineix el plànol del laberint
JocDelLaberint joc = new JocDelLaberint();

// Definim les fàbriques
FabricaDeLaberints fabrica_normal =
    new FabricaDeLaberints();
FabricaDeLaberints fabrica_encantada =
    new FabricaDeLaberintsEncantats();

// Obtenim laberints diferents
Laberint laberint_normal =
    joc.crearLaberint(fabrica_normal);
Laberint laberint_encantat =
    joc.crearLaberint(fabrica_encantada);
```

Prototype

Especifica els tipus d'objectes a crear mitjançant una **instància prototípica** i crea noves instàncies **copiant** aquest prototipus



Prototipus Declara una interfície per clonar-se

PrototipusConcret Implementa una operació per clonar-se

Client Crea nous objectes demanant a un prototipus que es cloni

Observacions

- Permet evitar la construcció d'una **jerarquia** paral·lela (com en el cas del *Mètode de Fabricació*) reduïnt així l'herència
- Al tenir que implementar l'operació `clonar()` hem de poder **modificar** les interfícies de les classes
- Quan les instàncies d'una classe poden tenir un d'entre uns pocs estats diferents, pot ser útil tenir un nombre equivalent de prototipus i clonar-los en comptes de crear manualment les instàncies amb l'estat adequat
- És possible crear nous prototipus en temps d'execució (*e.g.* objectes compostos que s'afegeixen al repertori)
- Nosaltres evitarem l'ús de la interfície de marcatge `Cloneable` definida al **Java SDK** (més informació a *Effective Java* de Joshua Bloch)

Problemes

Gestor de prototipus Quan el nombre de prototipus en el sistema no és fix, pot ser útil mantenir un registre dels prototipus existents. Els clients no gestionaran els prototipus sinó que els demanaran al gestor abans de clonar-los (o el gestor els hi retornarà els clons)

Implementació de clonar El principal problema consisteix en tractar objectes que continguin referències circulars.

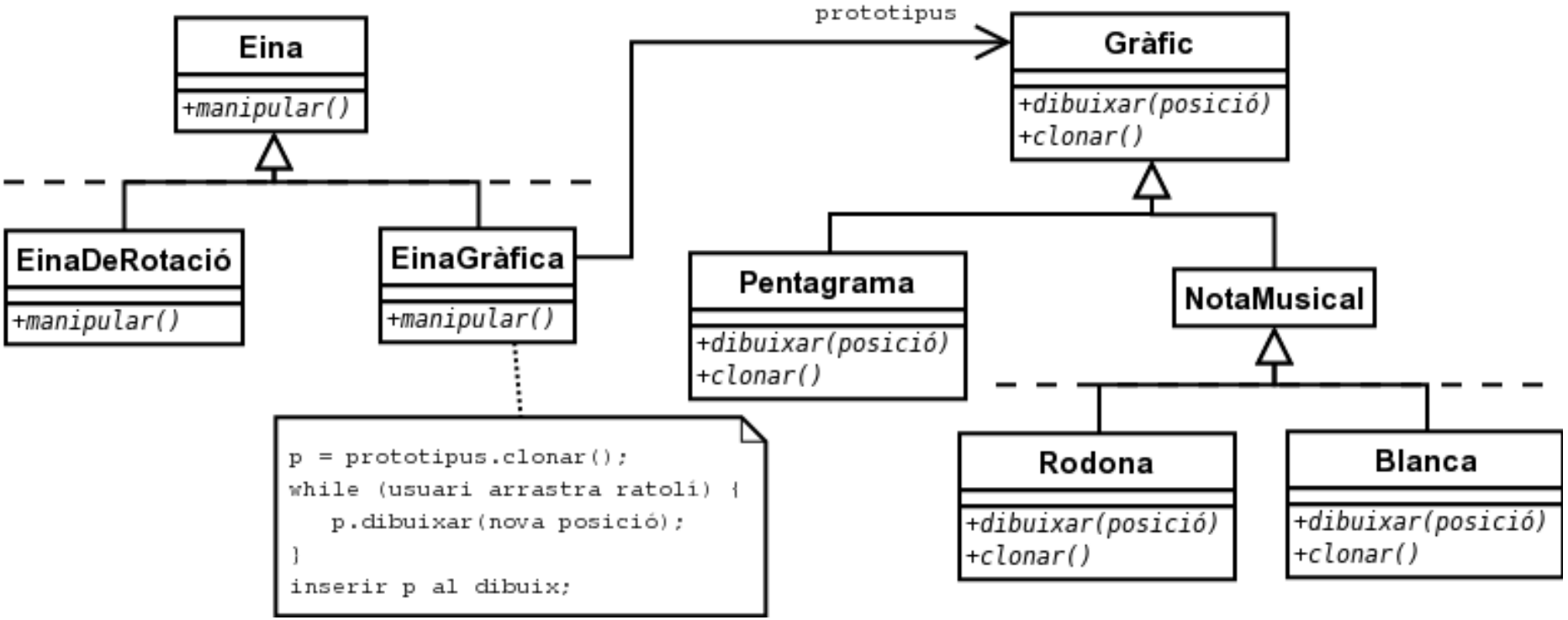
Còpia superficial És la forma més simple i sovint és suficient (*e.g.* constructor de còpia *per defecte* en C++)

Còpia profunda Sovint és necessària en el cas de prototipus complexes que han de funcionar de **forma independent**

Inicialització dels clons De vegades els clients voldran inicialitzar parts de l'estat intern dels clons (normalment no ho podrem fer a l'operació `clonar` doncs aquesta necessita una interfície uniforme)

- utilitzar operacions per modificar l'estat
- operacions d'inicialitzar

Solució exemple



FabricaDePrototipusLaberint

Aplicarem la idea dels **prototipus** a la **fàbrica de laberints** de manera que el constructor ens permeti variar els prototipus

```
class FabricaDePrototipusDeLaberint
  extends FabricaDeLaberints {
  private Laberint laberint;
  private Habitacio habitacio;
  private Paret paret;
  private Porta porta;
  public FabricaDePrototipusDeLaberint(
    Laberint laberint, Habitacio habitacio,
    Paret paret, Porta porta) {
      this.laberint = laberint;
      this.habitacio = habitacio;
      this.paret = paret;
      this.porta = porta;
    }
  .....
```

FabricaDePrototipusLaberint

```
    // continuació de FabricaDePrototipusLaberint
    public Paret ferParet() {
      return paret.clonar();
    }
    public Porta ferPorta(Habitacio h1, Habitacio h2) {
      Porta nova_porta = porta.clonar();
      nova_porta.inicialitzar(h1,h2);
      return nova_porta;
    }
    .......
}
```

Utilitzant FabricaDePrototipusLaberint

Podem usar `FabricaDePrototipusLaberint` per a crear una `FabricaDeLaberint` amb els tipus de components personalitzats

```
JocDelLaberint joc = new JocDelLaberint();

FabricaDePrototipus fabricaDeLaberintsSimples =
        new FabricaDePrototipusLaberint (new Laberint(),
                                         new Paret(),
                                         new Habitacio(),
                                         new Porta());

Laberint laberint =
    joc.crearLaberint(fabricaDeLaberintsSimples);
```

Implementant Porta

Posarem el codi de la classe `Porta` per mostrar l'operació `inicialitzar`

```
class Porta {
  private Habitacio habitacio1;
  private Habitacio habitacio2;
  public Porta() {}
  public Porta(Porta altra) {
    habitacio1 = altra.habitacio1;
    habitacio2 = altra.habiatcio2;
  }
  public void inicialitzar(Habitacio h1,
                           Habitacio h2) {
    habitacio1 = h1;
    habitacio2 = h2;
  }
  public Porta clonar() {
    return new Porta(this);
  }
}
```

Definint ParetExplosionada

La subclasse ParetExplosionada ha de redefinir clonar i implementar el seu constructor còpia

```
class ParetExplosionada extends Paret {
  private bool bomba;
  public ParetExplosionada() {...}
  public ParetExplosionada(ParetExplosionada altra) {
    super(altra);
    bomba = altra.bomba;
  }
  public ParetExplosionada clonar() {
    return new ParetExplosionada(this);
  }
}
```

Utilitzant les *explosions*

De forma equivalent definiríem i implementaríem `HabitacionsAmbBomba` i definiriem el laberint amb

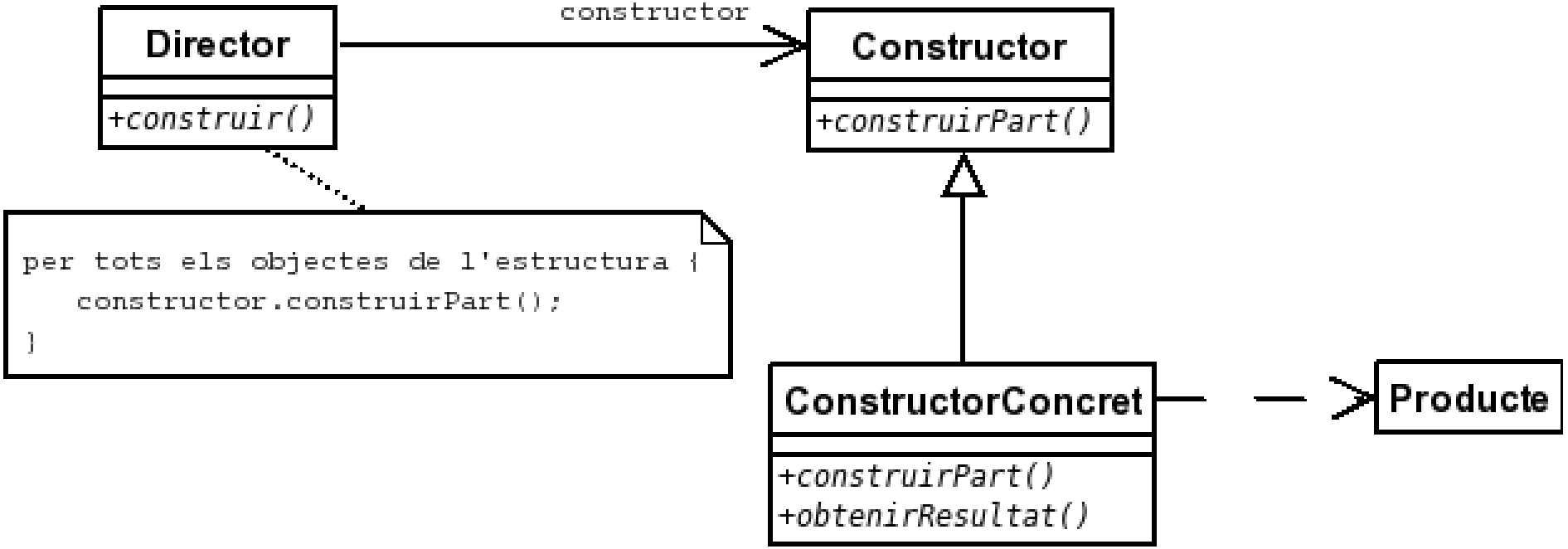
```
JocDelLaberint joc = new JocDelLaberint();

FabricaDeLaberints fabricaDeLaberintsAmbBomba =
  new FabricaDePrototipusLaberint(new Laberint(),
                                  new ParetExplosionada(),
                                  new HabitacioAmbBomba(),
                                  new Porta());

Laberint laberint =
  joc.crearLaberint(fabricaDeLaberintsAmbBomba);
```

Builder

Separa la construcció d'un objecte complex de la seva representació, de manera que el mateix procés de construcció pot crear diferents representacions



Participants

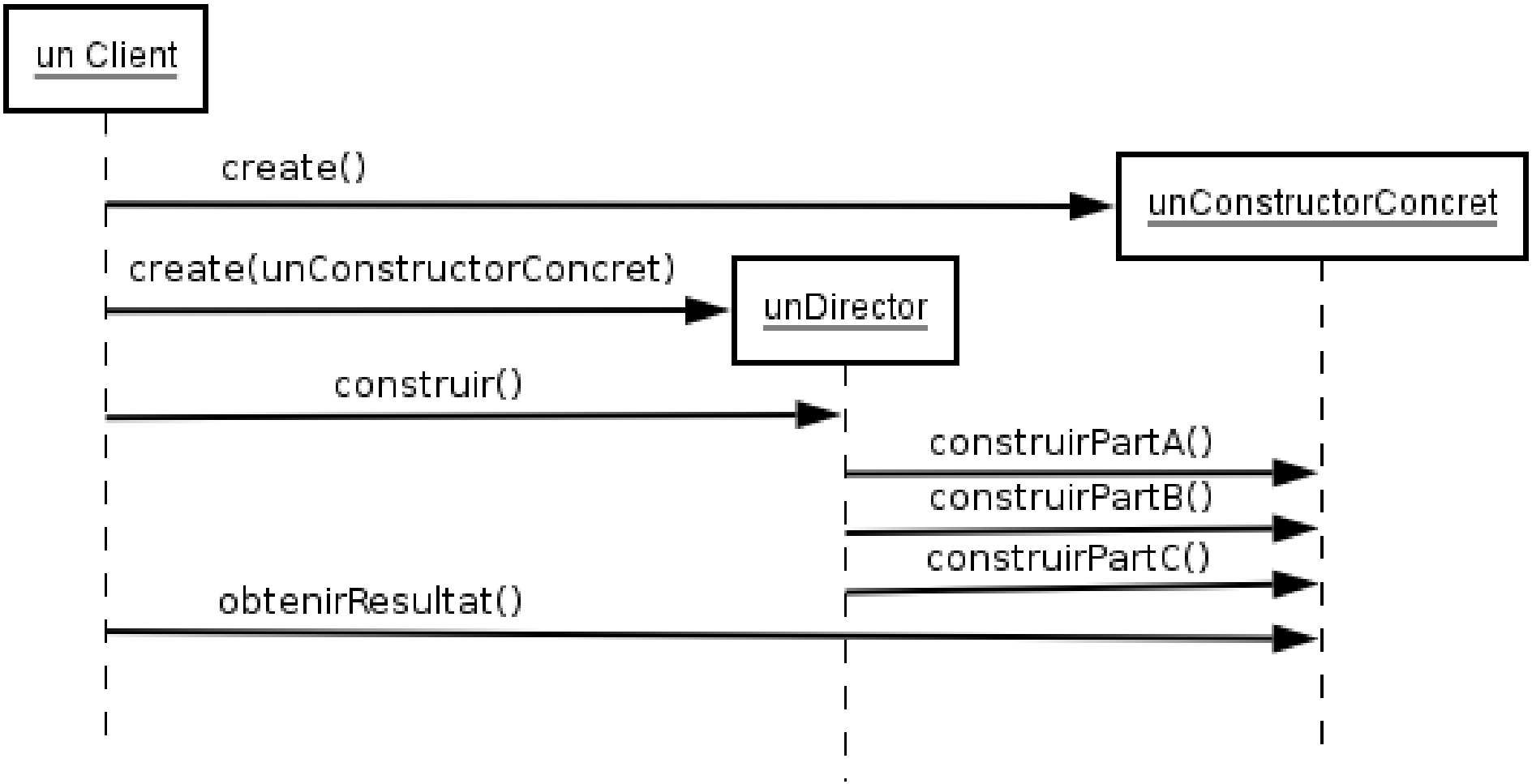
Constructor Defineix una interfície abstracta per a crear les parts d'un objecte `Producte`

ConstructorConcret Implementa la interfície `Constructor` per construir i assemblar les parts del producte; defineix la representació a crear; proporciona una interfície per a retornar el `Producte`

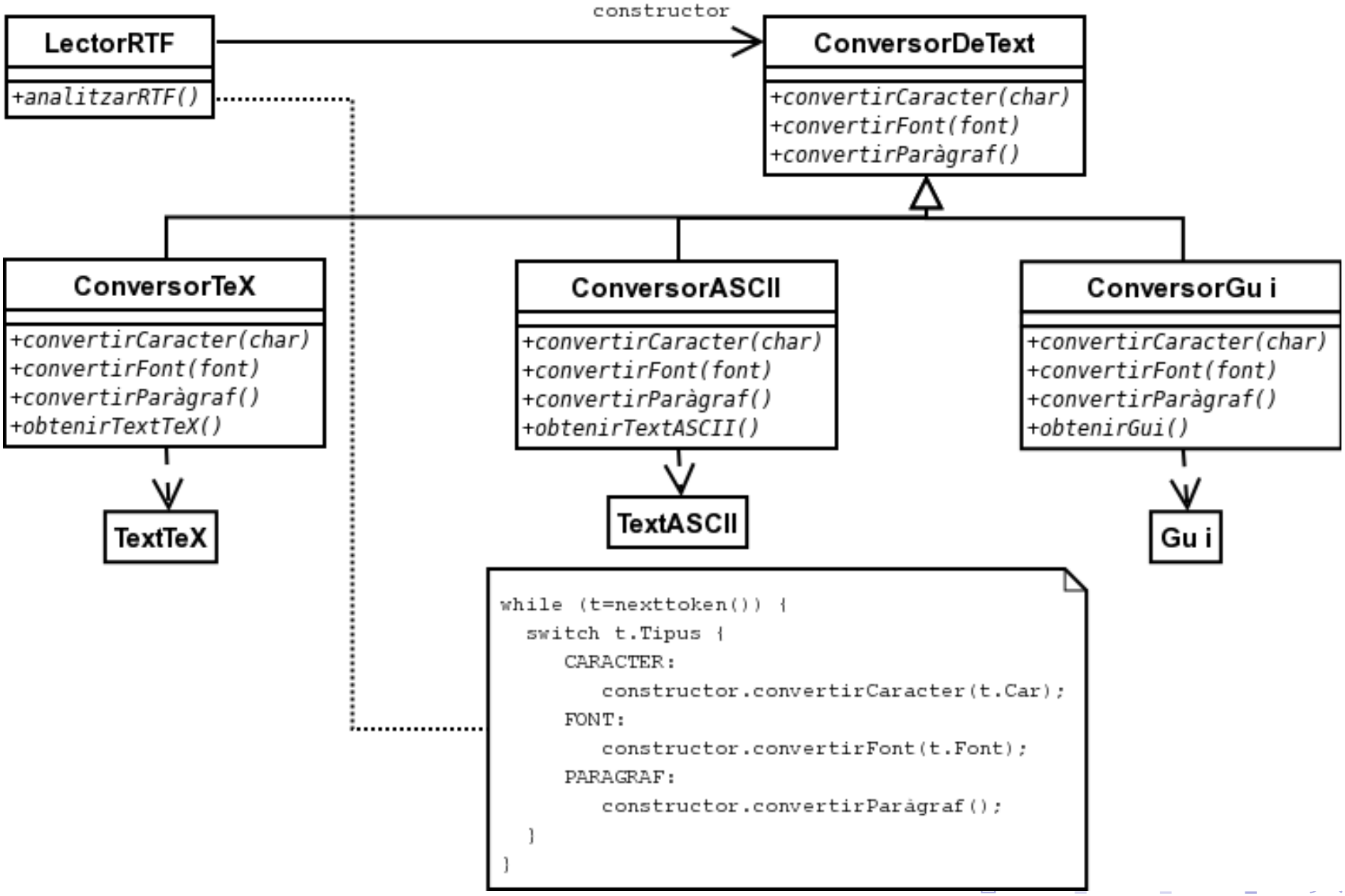
Director Construeix un objecte usant la interfície `Constructor`

Producte Representa l'objecte complex que es construeix; inclou les classes que defineixen les seves parts, incloent-hi interfícies per assemblar les parts al resultat final

Col·laboracions



Exemple: conversor formats



Observacions

- Permet variar la representació interna d'un producte ja que aquest es construeix via la interfície abstracta que proporciona `Constructor`
- Aïlla el codi de construcció i el de representació. Els clients no necessiten saber res de les classes que defineixen l'estructura interna dels productes que està construïnt
- Cada `ConstructorConcret` conté tot el codi per a crear i assemblar un determinat tipus de producte
- Proporciona un **control més fi** sobre el procés de construcció que els altres patrons de creació en els que el producte es construeix en un sol pas
  - El `Client` només obté el producte quan aquest està finalitzat
  - La interfície `Constructor` reflexa el procés de construcció del `Producte` i, per tant, permet aquest control més fi

Aspectes d'implementació

Interfície d'assemblatge i construcció La interfície de la classe `Constructor` ha de ser suficientment general per a construir els productes de tots els `ConstructorConcrets`

- Normalment és suficient amb un procés que va afegint les noves parts al producte
- Un cas diferent és quan cal accedir a part del producte construït
- Un altre cas són les estructures que es creen *bottom-up*

Classes abstractes pels productes En general els productes poden ser tan diferents que és difícil una **classe pare comuna**. Com el client sol configurar el `Director` amb el `ConstructorConcret` ja coneix el tipus de producte que està construïnt

Mètodes buïts al Constructor En Java els mètodes no es declaren com abstractes sinó amb implementació buïda ja que així els `ConstructorConcrets` només han de redefinir les parts en les que estan interessats

Definint ConstructorLaberint

Definirem una versió de `crearLaberint` que rebi com a paràmetre un objecte `ConstructorLaberint`

```
abstract class ConstructorLaberint {
  public void inicialitzarLaberint() {}
  public void afegirHabitacio(int n) {}
  public void afegirPorta(int habitacio_origen,
                          int habitacio_desti
                          Direccion dir) {}
}
```

- Aquesta classe pot crear tres coses: el laberint, habitacions i portes
- Les subclasses de `ConstructorLaberint` redefiniran aquestes operació per a fer diferents laberints

Modificant crearLaberint

```
class JocDelLaberint {
  public void crearLaberint(ConstructorLaberint constructor) {
    constructor.inicialitzarLaberint();
    constructor.afegirHabitacio(1);
    constructor.afegirHabitacio(2);
    constructor.afegirPorta(1,2,Est);
  }
}
```

- L'objecte `constructor` oculta la **representació interna** de `Laberint` (les classes que defineixen habitacions, portes, ...) i com s'**assemblen** entre si
- Hem eliminat la construcció de les parets a la interfície del constructor doncs això **simplifica** la creació de laberints (la interfície del `constructor` aïlla de la representació dels `Productes`

ConstructorLaberintEstàndar

```
class ConstructorLaberintEstandar
      extends ConstructorLaberint {
  private Laberint laberint_actual;
  private Direccio direccioContraria(Direccio d) {
    ....
  }
  public void inicialitzarLaberint() {
    laberint_actual = new Laberint();
  }
  public void afegirHabitacio(int num) {
    if (!laberint_actual.conteHabitacio(n)) {
      Habitacio habitacio = new Habitacio(n);
      laberint_actual.afegirHabitacio(habitacio);
      habitacio.establirCostat(NORD, new Paret());
      habitacio.establirCostat(SUD,  new Paret());
      habitacio.establirCostat(EST,  new Paret());
      habitacio.establirCostat(OEST, new Paret());
    }
  }
  .....
```

ConstructorLaberintEstandar

```
  // continuació de ConstructorLaberintEstandar
  public void afegirPorta(int n1, int n2, Direccio dir) {
    Habitacio h1 = laberint_actual.obtenirHabitacio(n1);
    Habitacio h2 = laberint_actual.obtenirHabitacio(n2);
    if ( h1!=null && h2!=null) {
      Porta p = new Porta(h1,h2);
      h1.establirCostat(dir, p);
      h2.establirCostat(direccioContraria(dir),p);
    }
  public Laberint obtenirLaberint() {
    return laberint_actual;
  }
}
```

Ara els clients el poden utilitzar per a crear un laberint

```
JocDelLaberint joc = new JocDelLaberint();
ConstructorLaberintEstandar constructor =
                    new ConstructorLaberintEstandar();
joc.crearLaberint(constructor);
Laberint laberint = constructor.obtenirLaberint();
```

Aprofitant la flexibilitat

Podem definir un `ConstructorLaberint` més exòtic que, en comptes de crear el `Laberint`, només **compti** el nombre d'elements que s'han utilitzat.

```
class ConstructorLaberintComptador
      extends ConstructorLaberint {
  private int num_portes; private int num_habitacions;
  public void inicialitzarLaberint() {
    num_portes = 0; num_habitacions = 0;
  }
  public void afegirHabitacio(int) {
    ++num_habitacions;
  }
  public void afegirPorta(int h1, int h2, Direccio d) {
    ++num_portes;
  }
  public Parella obtenirCompteig() {
    return new Parella(num_habitacions, num_portes);
  }
}
```

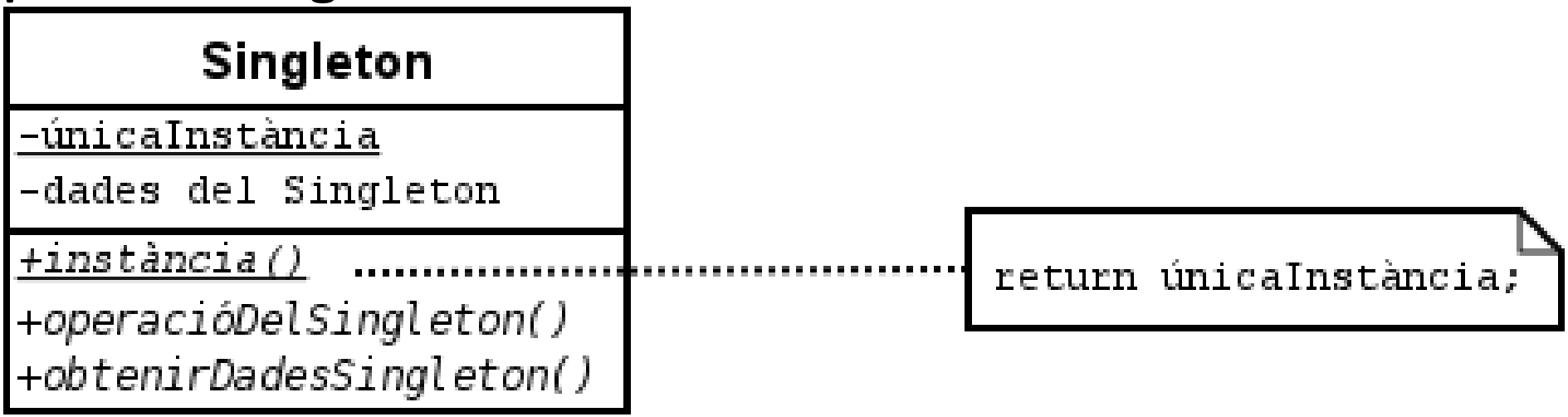
on `Parella` és una classe auxiliar per a guardar un parell d'enters.

Utilitzant ConstructorLaberintComptador

```
JocDelLaberint joc = new JocDelLaberint();
ConstructorLaberintComptador constructor =
                    new ConstructorLaberintComptador();
joc.crearLaberint(constructor);
Parella parella = constructor.obtenirCompteig();
System.out.print("El Laberint té ");
System.out.print(parella.getFirst());
System.out.print(" habitacions i ");
System.out.print(parella.getSecond());
System.out.println(" portes.");
```

Singleton

Garantitza que una classe **només** tingui una instància i proporciona un **punt d'accés global** a ella



- L'operació `instancia` permet als clients accedir a la única instància de la classe
- És una **operació de classe** (estàtica) i no una operació d'**instància**
- Aquesta operació sol ser responsable de **crear** aquesta única instància

Observacions

El patró **singleton** proporciona els següents *beneficis*:

Accés controlat a l'única instància() Podem controlar com s'accedeix a aquesta única instància ja que **només** es pot fer via `instancia`

Disminució de l'espai de noms Ja no calen variables globals per accedir a objectes globals

Permet un nombre variable d'instàncies Podem modificar el patró per a permetre-ho (només cal canviar la operació d'accés a la instància)

Implementació

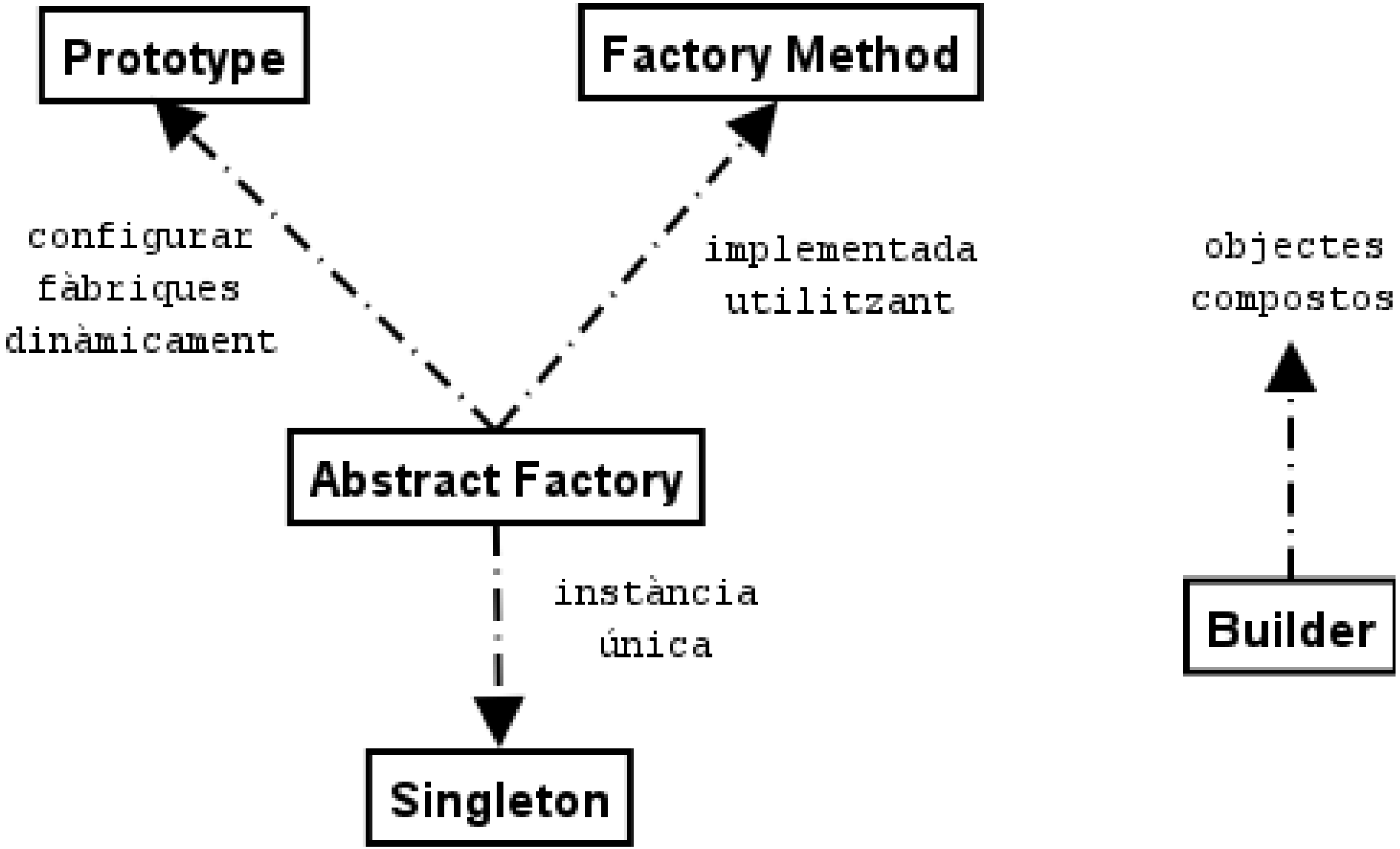
La classe es declara com:

```
class Singleton {
  private static Singleton instance;
  protected Singleton() {...}
  public static Singleton instancia() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Aplicació a FabricaDeLaberints

```
class FabricaDeLaberints {
  private static FabricaDeLaberints fabrica;
  protected FabricaDeLaberints() {}
  public static FabricaDeLaberints instancia() {
    if (fabrica == null)
        fabrica = new FabricaDeLaberints;
    return fabrica;
  }
  public Laberint ferLaberint() {...}
  // ... resta de la interfície
}
```

Relacions entre el patrons



Bibliografia

- Capítol 3 del llibre *Patrones de diseño*.

Llicència