Avantatges de la utilització dels llenguatges orientats a objectes en el disseny d'eines de CAD per a electrònica de potència

Àlex Méndez, Josep Bordonau i Joan Peracaula

Els llenguatges orientats a objectes s'estan emprant en la implementació de programes de CAD perquè permeten resoldre problemes molt complexos. En aquest document es presenten raons addicionals per a la seva utilització en el disseny d'eines de CAD per a Electrònica de Potència.

1. INTRODUCCIÓ.

Normalment, tot el software que realitza tasques de CAD té una complexitat elevada. És per aquesta raó fonamentalment, que moltes de les aplicacions de CAD que s'estan desenvolupant actualment ho fan en algun llenguatge orientat a objectes[1]. El fet que els llenguatges de procediments, també anomenats darrerament orientats a procediments, no siguin aconsellables per a afrontar problemes molt complexos es deu bàsicament a que utilitzen un model de capsa negra, que no impedeix els efectes laterals i que tampoc obliga a un desenvolupament massa estructurat. Els llenguatges orientats a objectes resolen molts d'aquests problemes permetent la realització d'un software molt més complexe, robust i de manteniment senzill[2].

"Als llenguatges orientats a objectes el programador pot definir dades abstractes."

Com que aquesta és la raó fonamental per a la seva utilització, molta de la gent que és aliena al seu ús, i fins i tot programadors novells en aquest camp, pensen que és l'única aportació d'aquest tipus de llenguatges. De fet, és un error lògic si tenim en compte que els temps d'aprenentatge i complexitat d'aquests tipus de llenguatges són majors que els llenguatges funcionals, i que per tant si alguna cosa cal destacar-ne és la raó bàsica de desenvolupament. A més, quan es van desenvolupar, aquest era l'objectiu d'aquests llenguatges. Tanmateix, si es segueix el procés evolutiu de totes les implementacions d'aquest tipus de llenguatges, es comprova que s'hi han anat afegint característiques que els converteixen en eines força potents i que poden ésser aprofitades per a simplificar i fer més potent el software desenvolupat. De fet aquestes característiques fan que molts algorismes es puguin plantejar d'un mode molt més general i potent. En aquest document es mostren els avantatges que aquest tipus de llenguatges poden oferir fàcilment en el disseny d'eines de CAD per a Electrònica de Potència sobre els llenguatges orientats a procediments. Aquests avantatges s'il·lustren amb exemples senzills implementats en C++.

ÀLEX MÉNDEZ FERRÉS *és enginyer industrial. Participa en diversos projectes de recerca en el departament d'Enginyeria Electrònica de la UPC. Actualment gaudeix d'una beca de F.I. de la Generalitat de Catalunya.*
JOSEP BORDONAU FARRERONS *és professor Titular del departament d'Enginyeria Electrònica de la UPC des de 1990. Actualment és sots-director de recerca del Deptartament d'Enginyeria Electrònica de la UPC.*
JOAN PERACAULA ROURA *és catedràtic del departament d'Enginyeria Electrònica de la UPC des de 1971. Actualment és responsable del programa de doctorat d'aquest departament.*

2. DADES, CLASSES I OBJECTES.

En un llenguatge de programació hi ha només un tipus de dades: les

incorporades. Les incorporades són aquelles que el compilador reconeix com a «seves». En el cas del C són les típiques: char, int, float i double. Una de les característiques dels llenguatges orientats a objectes és que el programador defineix dades abstractes, que són les dades que com a unitat tenen sentit dins d'un programa[1]. Però com que les dades porten sempre associades operacions i funcions pròpies cal que es defineixin aquestes operacions i funcions intrínseques a les dades mateixes, sabent així el compilador què fer quan es troba amb una dada abstracta. Així es pot pensar una classe com una definició de nova dada, un paquet on hi van dades, operacions i funcions associades i els objectes com les realitzacions en el programa de les classes. Per exemple, en una eina de CAD les dades poden ésser components i definir com a operacions l'associació en sèrie, l'associació en paral·lel, etc.

Tot seguit es mostra part de la definició d'una classe de components simplificada, on com a dades hi hauria l'Origen del component (si és un component o una agrupació d'uns quants), el nombre de components que el formen, la impedància equivalent, i el pes del conjunt. Com a funcions associades es defineix posar en sèrie i posar en paral·lel:

```
class Component
{
  ....
  // dades membre
  Linia Origen;
  int NombreComponents;
  complex Z;
  double Pes;

  //funcions membre
  Component();
  Component(Component &);
  ~Component();
  int TipusComponent();
  virtual char * NomComponent();
  virtual void Llegir(char * Nom);
  virtual void Escriure(char * Nom);
  Component operator=(Component);
  Component operator+(Component);
  Component operator||(Component);
  ....
};
```

Un cop definides les noves dades que es faran servir en un programa i les operacions i funcions associades, el programador pot pensar en components electrònics en comptes d'estructures o floats, i pot pensar en posar en sèrie o paral·lel en lloc de cridar la funció A o la funció B. Un cop definides les dades i funcions associades, i un cop depurades, es pot desenvolupar el procés de CAD en un nivell tan elevat com es vulgui. Aquesta estructuració és la que permet una major complexitat en aquests programes, i com ja s'ha dit és l'avantatge més conegut. Però n'hi ha d'altres, menys coneguts, i potser tant interessants per a un dissenyador d'eines de CAD.

3. HERÈNCIA.

Si vostè és un programador de llenguatges funcionals segurament estarà pensant que tot això potser sí sigui més estructurat, però molt més laboriós a causa de què cal definir moltes dades i moltes funcions[3]. Però això no és cert, ja que hi ha un mecanisme que ha anat evolucionant fins a ésser molt potent: l'herència.

L'herència permet definir una nova dada, com un conjunt de dades abstractes i dades incorporades, tenint la possibilitat de definir noves dades i nous procediments damunt les dades abstractes ja definides. Tornant a l'exemple anterior, suposem que es vol definir una dada que sigui una resistència. És clar que una resistència és un component i que pot compartir la majoria de les dades i funcions d'un component general. Per tant, per a definir una resistència, només cal dir que és un component, i què té de nou i/o diferent d'un component:

```
//-------------------------
class Resistor: public Component
{
 public:
   Resistor(Component & C);
   ~Resistor();
          virtual char *
NomComponent();
   void FuncioA(){}
};
```

En el cas de l'exemple per a definir una resistència nomes cal que és un component, que no té noves dades i que dues funcions més asociades, una de les quals substitueix a una de les que tindria pel fet d'ésser un component.

Les herències poden ésser de dos tipus: per inclusió (quan una de les dades de la nova classe és una dada abstracta) i per evolució (quan una classe inclou les dades de la classe progenitora i afegeix noves funcions o anul·la d'altres). les herències d'evolució poden ésser simples, múltiples, d'expansió, de restricció, etc.

Queda clar, que els diferents tipus d'herència permeten definicions molt ràpides i força complexes de les dades que es necessiten . Aquesta característica permet la definició de dades més sofisticades, de dades millorades, de dades noves a partir de les anteriors. Tot això suposa:

- Reutilització automàtica i molt gran del codi ja escrit. Les reutilitzacions són del 90% del codi que es va desenvolupant, i d'una forma gairebé inconscient.

- Aprenentatge per part del programador sobre quins són els objectes més adients. Com que es pot anar reescribint damunt les classes anteriors i fent-ne de noves que afegeixen coses o en treuen, amb poques línies, queda clar que es poden anar desenvolupant les eines sobre la marxa i començar amb classes poc definides i anar complentant amb els successius intents.

- La major part del temps es dedica a pensar en termes d'enginyeria i no en termes informàtics, un cop que es tenen les classes bàsiques i que es vol desenvolupar la lògica del programa. Fins i tot es pot pensar en dos nivells de programadors: un que desenvolupi les classes base i

un programador usuari que utilitzi les classes base. Això de fet ja es produeix comercialment de formes més o menys clares. Suposem per exemple que es tinguin les classes corresponents a transformadors. Llavors fer un programa que calculi l'equivalent elèctronic a una associació de transformadors, pot ésser tant senzill com:

```
#include <trafos.hpp>
#include «basics.hpp»

void main()
{
  Transformador
     Trafo1(«T1.DAT»),
     Trafo2(«T2.DAT»),
     Trafo3(«T3.DAT»),
     Trafo4(NULL);

  Trafo4 = (Trafo1 +Trafo2 ) || Trafo3;

  cout << Trafo4.DadesElectriques();
}
```

Un altre aspecte important del sistema d'herències és que el coneixement sobre les dades que es volen estructurar en classes serveix per a fer-ne una bona classificació en famílies. És a dir, per determinar les herències cal fer cas dels coneixements d'Enginyeria Electrònica i no pas dels d'Informàtica. En resum, les intuïcions i coneixements sobre els circuits electrònics són els que han de guiar al programador per tal de dirigir el disseny de les eines informàtiques.

4. POLIMORFISME I CONTENI-DORS.

Si vostè és un programador de llenguatges funcionals segurament estarà pensant que tot això ja s'ho imaginava, ja que tot el que de moment s'ha explicat són millores notacionals, estructurals i metodològiques. Només aquestes millores ja en justificarien l'ús. Però n'hi ha que no tenen aquest caire formal i que permenten donar un enfocament més avançat als algorismes. Una d'aquestes característiques que van més enllà d'aspectes formals és el polimorfisme[4].

El polimorfisme és una propietat que poden tenir les classes si el programador així ho decideix. És una propietat més aviat complexe a causa de què va combinada amb el tipus d'herència i que, en definitiva, permet diversos tipus d'objectes i classes dins les mateixes famílies de classes. Com que l'objecte d'aquest document es redueix a mostrar els avantatges dels llenguatges orientats a objectes, només es destacaran certs aspectes del polimorfisme sense definir-lo.

"Els diferents tipus d'herències permeten definicions molt ràpides i força complexes de les dades que es necessiten."

Un dels aspectes que més interessen del polimorfisme en aquest camp és el fet que, mitjantçant el polimorfisme, es poden definir operacions, funcions i algorismes que processin les classes pares, sense saber com seran els descendents d'aquestes classes, i que, tanmateix, les funcions així definides també siguin vàlides per als objectes de les classes descendents[2]. Com a exemple suposem una classe que faci una llista de components electrònics. La llista de components pot tractar objectes de les classes resistència, inductor i capacitor malgrat que hagi estat escrita per a objectes que pertanyen a la classe de components, ja que quan es va definir la classe de components, no se sabia com serien les classes resistència, condensador, etc. Això permet anar sofisticant els objectes sense que es tingui que reescriure una sola línia vàlida pels objectes anteriors. Però a més a més, permet que un mateix algorisme pugui manegar dades diferents i continuï essent eficaç, tot i que aquest algorisme, quan es va escriure, no sabia les dades que processaria. Exemple:

```
//xxxxxxxxxxxxxxxxxxxxxxxxxxx
class GestorElements
{
 public:
  int
   NombreElements;
  Component
    * LlistaComponents[MAXCOMP];

  GestorElements();
  ~GestorElements();
  virtual void Agregar(Component & C);
  virtual void Ser(Component & C);
  virtual void Par(Component & C);
                virtual void
  Llistar();
  };
```

La classe GestorElements pot gestionar elements derivats de la classe Component, sense tenir coneixement de com estan heretats de Components. D'un objecte que pertany a una classe que conté objectes polimòrfics s'anomena contenidor. Aquest concepte s'emprarà més endavant. La classe GestorElements és una classe contenidor ja que conté objectes d'altres classes, de forma totalment dinàmica, ja que a priori no es coneixen ni quants ni quins.

Un altre exemple concret és el següent: suposi que desitja crear un algorisme que trobi una combinació de transistor-snuber que sigui l'adient. El mateix algorisme pot servir i executar-se alhora amb diferents tipus de transistors (Mosfets, Bipolars, etc) i diferents tipus de snubers (RC, Diode+RC, etc) encara que cada tipus tingui diferents dades i diferents funcions associades.

Un altre aspecte del polimorfisme molt apreciat en el disseny d'eines de CAD és el fet que en temps d'execució es poden «crear» objectes polimòrfics com a conseqüència de les opcions escollides per l'usuari o com a resultat del procés de les dades. Un exemple és una classe de circuits, que pot acabar tenint diferent nombre i diferents tipus de components, sense que el programador ho hagi previst ni imaginat, i que el programa funcioni correctament. Exemple: suposem una eina dissenyadora de snubers. A priori no es coneix quin tipus de xarxa

tindrà, i quan es conegui tampoc quins seran els components que l'integraran. Tant la xarxa com els components es poden determinar de forma dinàmica, partint d'unes xarxes ja conegudes i d'uns components disponibles, i finalitzant en un tipus de xarxa de snuber que no sigui cap de les de partida (a partir de les existents en que dissenyada una nova) i que alguns dels components siguin combinacions dels de partida per aconseguir funcions d'impedància millors.

5. CREACIÓ DINÀMICA D'OBJECTES.

Un dels fets més habituals en un sistema de CAD és que el programador no sap quines són les dades que es crearan durant l'execució del programa. És el mateix programa el que s'encarrega d'anar creant les dades a mida que són necessàries. Aquesta possibilitat ja estava prevista en llenguatges com el C. Però és en els llenguatges orientats a objectes quan adquireix més sentit, ja que durant l'execució del programa es determinen quins són els tipus de dades i quines dades en concret calen. Jugant amb el polimorfisme fins i tot és relativament fàcil el fet d'aconseguir la creació de tipus d'objectes polimòrfics nous, i que tots els procediments bàsics del sistema de CAD els puguin tractar com si de fet ja fossin coneguts[3].

Com a exemple, la classe gestora de la llista de components anteriorment esmentada crea els objectes que resulten de les accions de l'usuari, sense que abans de l'execució del programa es conegui quants objectes seran creats i de quin tipus:

```
//-------------------------------------

void GestorElements::Agregar(Component & C)
{
 if(NombreElements>=MAXCOMP)
  {
     cout << «\nATENCIO: No és possible»
          <<«agregar més elements a la llista\n»;
   return;
  }

 switch(C.TipusComponent())
 {
  case RESISTOR:
 LlistaComponents[NombreElements++]
            = new Resistor(C);
  break;
  case CAPACITOR:
 LlistaComponents[NombreElements++]
        = new Capacitor(C);
  break;
  case INDUCTOR:
 LlistaComponents[NombreElements++]
        = new Inductor(C);
  break;
 }
}
```

Com es pot veure el polimorfisme dóna molt més sentit a la creació dinàmica d'objectes, ja que es poden crear els algorismes que permetin la generació en temps d'execució i com a resultat de la interacció entre l'usuari i el Sistema d'Eines de CAD de nous components o circuits, que el propi programador no havia imaginat, i que malgrat tot, el programa que ha realitzat segueix processant correctament aquestes noves dades.

6. PERSISTÈNCIA.

Dels punts anteriors queda clar que cal dotar d'un mecanisme automàtic els objectes generats durant una execució perquè puguin sobreviure el temps que duri el programa. Aquest emmagatzament de tots els objectes generats durant les execucions i del seu estat (el contingut de les dades) permet al propi programa evaluar els millors camins, les preferències dels usuaris i els rendiments de les eines. Permet informar al programador de posibles defectes i dóna experiència al programa.

En resum, la persistència tal i com es planteja en aquest document, permet disposar d'un mode d'aprenentatge pel programador i per les pròpies eines de CAD. Malauradament, aquest aspecte no ha estat encara gaire desenvolupat ni valorat en les implementacions actuals dels llenguatges oirentats a objectes pel que cal dotar de forma manual d'aquest aspecte al software desenvolupat. Però ja que és, o hauria d'ésser una característica més, resulta força natural la seva implementació dins del sistema de classes a partir del sistema d'herències.

7. CONCLUSIONS.

En aquest document han estat presentats els avantatges dels llenguatges orientats a objectes sobre els llenguatges orientats a procediments en el disseny d'eines de CAD per a circuits electrònics de potència:

1.- El sistema de classes i herències permet una complexitat major en el codi desenvolupat, una major reutilització del codi i una major comoditat i eficàcia en el disseny del codi. També permet l'aprenentatge del programador sobre els objectes que està dissenyant, de manera que pugui progressar de forma continuada.

2.- El polimorfisme permet la creació d'algorismes i funcions sense haver de conèixer les dades que es faran servir. Això permet treballar sense gaire esforç amb diferents components o circuits, variant els paràmetres o les topologies, sense variar els algorismes o funcions.

3.- La creació dinàmica d'objectes juntament amb el polimorfisme, permet el disseny de noves topologies o agrupacions de components a partir dels ja existents, i verificació de resultats i anàlisi de tipus evolutiu.

4.- La persistència dels objectes creats dinàmicament durant les execucions dels programes, doten de les dades que calen en un sistema amb aprenentatge.

8. REFERÈNCIES.

[1] G. Booch: *Object-Oriented Design with Applications*, Benjamin/Cummings Publishing Co. Inc, 1991.

[2] B. Cox: *Object-Oriented Programming, an Evolutionary Approach.*, Addison Wesley, 1987.

[3] B. Meyer: *Object-Oriented Software Construction*, Prentice Hall, 1988.

[4] B. Stroustrup: *The C++ Programming Language*, Addison Wesley, 2ª edición, 1991.