Paradigmas Fundamentales de Programación

Programación declarativa vs. Programación con estado

Juan Francisco Díaz Frias

Maestría en Ingeniería, Énfasis en Ingeniería de Sistemas y Computación
Escuela de Ingeniería de Sistemas y Computación,
home page: http://eisc.univalle.edu.co
Universidad del Valle - Cali, Colombia

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Universidad del Valle

Plan

Universidad del Valle

Plan

Colecciones con estado (1)

Hasta ahora, colecciones sin estado:

- Tuplas
- Registros

Arreglos

- Asociación de enteros con valores parciales.
- Dominio: enteros consecutivos entre un límite inferior y un límite superior.
- Acceso y modificación de un elemento en tiempo constante.
- Mozart provee los arreglos como un TAD predefinido en el módulo `Array`.

Operaciones sobre arreglos

- `A={NewArray L H I}`
- `{Array.put A I X}`: lo mismo que `A.I:=X`.
- `X={Array.get A I}`: lo mismo que `X=A.I`.
- `L={Array.low A}`: índice inferior de `A`.
- `H={Array.high A}`: índice superior de `A`.
- `R={Array.toRecord L A}`
- `A={Tuple.toArray T}`
- `A2={Array.clone A}`

Colecciones con estado (1)

Hasta ahora, colecciones sin estado:

- Tuplas
- Registros

Arreglos

- Asociación de enteros con valores parciales.
- Dominio: enteros consecutivos entre un límite inferior y un límite superior.
- Acceso y modificación de un elemento en tiempo constante.
- Mozart provee los arreglos como un TAD predefinido en el módulo `Array`.

Operaciones sobre arreglos

- `A={NewArray L H I}`
- `{Array.put A I X}`: lo mismo que `A.I:=X`.
- `X={Array.get A I}`: lo mismo que `X=A.I`.
- `L={Array.low A}`: índice inferior de `A`.
- `H={Array.high A}`: índice superior de `A`.
- `R={Array.toRecord L A}`
- `A={Tuple.toArray T}`
- `A2={Array.clone A}`

Colecciones con estado (1)

Hasta ahora, colecciones sin estado:

- Tuplas
- Registros

Arreglos

- Asociación de enteros con valores parciales.
- Dominio: enteros consecutivos entre un límite inferior y un límite superior.
- Acceso y modificación de un elemento en tiempo constante.
- Mozart provee los arreglos como un TAD predefinido en el módulo `Array`.

Operaciones sobre arreglos

- `A={NewArray L H I}`
- `{Array.put A I X}`: lo mismo que `A.I:=X`.
- `X={Array.get A I}`: lo mismo que `X=A.I`.
- `L={Array.low A}`: índice inferior de `A`.
- `H={Array.high A}`: índice superior de `A`.
- `R={Array.toRecord L A}`
- `A={Tuple.toArray T}`
- `A2={Array.clone A}`

Colecciones con estado (2)

Diccionarios

- Asociación de constantes sencillas (átomos, nombres, o enteros) con valores parciales.
- Tanto el dominio como el rango de la asociación se pueden cambiar.
- Provee operaciones para consultar, modificar, agregar o eliminar ítems durante la ejecución.
- Operaciones eficientes: consultar y modificar se hacen en tiempo constante, y agregar/eliminar se hace en tiempo constante amortizado.
- La memoria activa requerida es proporcional al número de ítems en la asociación.
- El sistema Mozart provee diccionarios como un TAD en el módulo `Dictionary`.

Operaciones sobre diccionarios

- `D={NewDictionary}`: diccionario vacío nuevo.
- `{Dictionary.put D LI X}`: lo mismo que `D.LI:=X`.
- `X={Dictionary.get D LI}`: lo mismo que `X=D.LI`.
- `X={Dictionary.condGet D LI Y}`: variación de get. Devuelve `Y` si `D.LI` no existe.
- `{Dictionary.remove D LI}`.
- `{Dictionary.member D LI B}`.
- `R={Dictionary.toRecord L D}`.
- `D={Record.toDictionary R}`.
- `D2={Dictionary.clone D}`.

Colecciones con estado (2)

Diccionarios

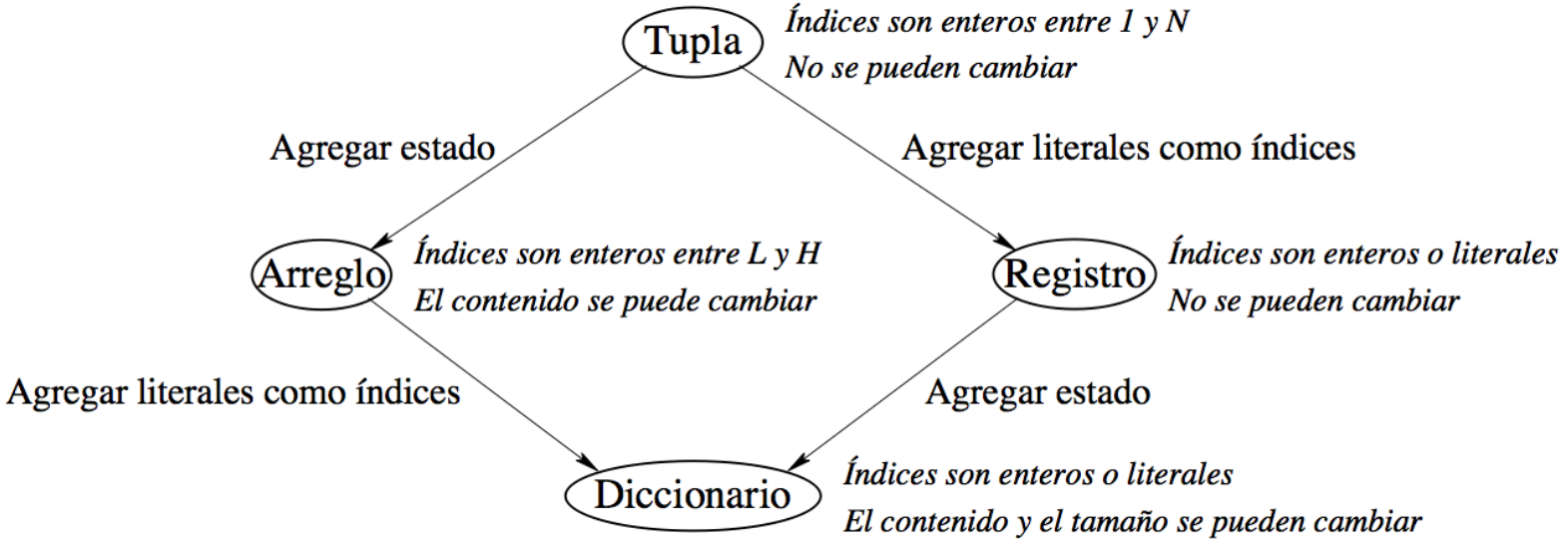
- Asociación de constantes sencillas (átomos, nombres, o enteros) con valores parciales.
- Tanto el dominio como el rango de la asociación se pueden cambiar.
- Provee operaciones para consultar, modificar, agregar o eliminar ítems durante la ejecución.
- Operaciones eficientes: consultar y modificar se hacen en tiempo constante, y agregar/eliminar se hace en tiempo constante amortizado.
- La memoria activa requerida es proporcional al número de ítems en la asociación.
- El sistema Mozart provee diccionarios como un TAD en el módulo `Dictionary`.

Operaciones sobre diccionarios

- `D={NewDictionary}`: diccionario vacío nuevo.
- `{Dictionary.put D LI X}`: lo mismo que `D.LI:=X`.
- `X={Dictionary.get D LI}`: lo mismo que `X=D.LI`.
- `X={Dictionary.condGet D LI Y}`: variación de get. Devuelve `Y` si `D.LI` no existe.
- `{Dictionary.remove D LI}`.
- `{Dictionary.member D LI B}`.
- `R={Dictionary.toRecord L D}`.
- `D={Record.toDictionary R}`.
- `D2={Dictionary.clone D}`.

Colecciones con estado (3)

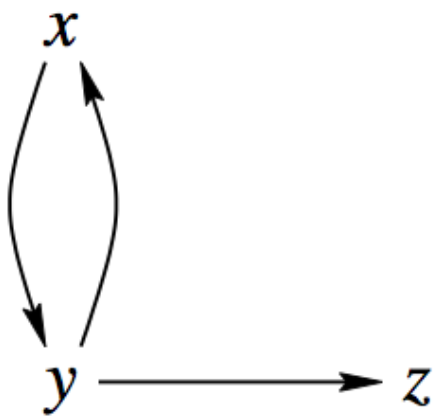
Comparación colecciones con estado y sin estado

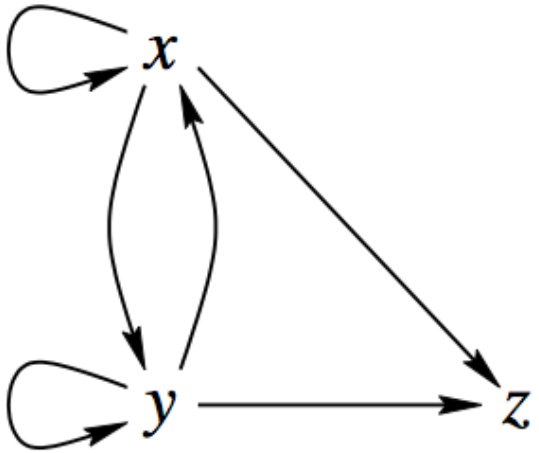


Plan

La clausura transitiva de un grafo

- Grafo dirigido: $G = (V, E)$, dados $x, y \in V$(nodos), $(x, y) \in E$ ssi hay una arista de x a y.
- La clausura transitiva de G, es un grafo dirigido nuevo, con los mismos nodos, y que tiene una arista entre dos nodos siempre que el grafo original tenga un camino entre esos mismos dos nodos.





Universidad del Valle

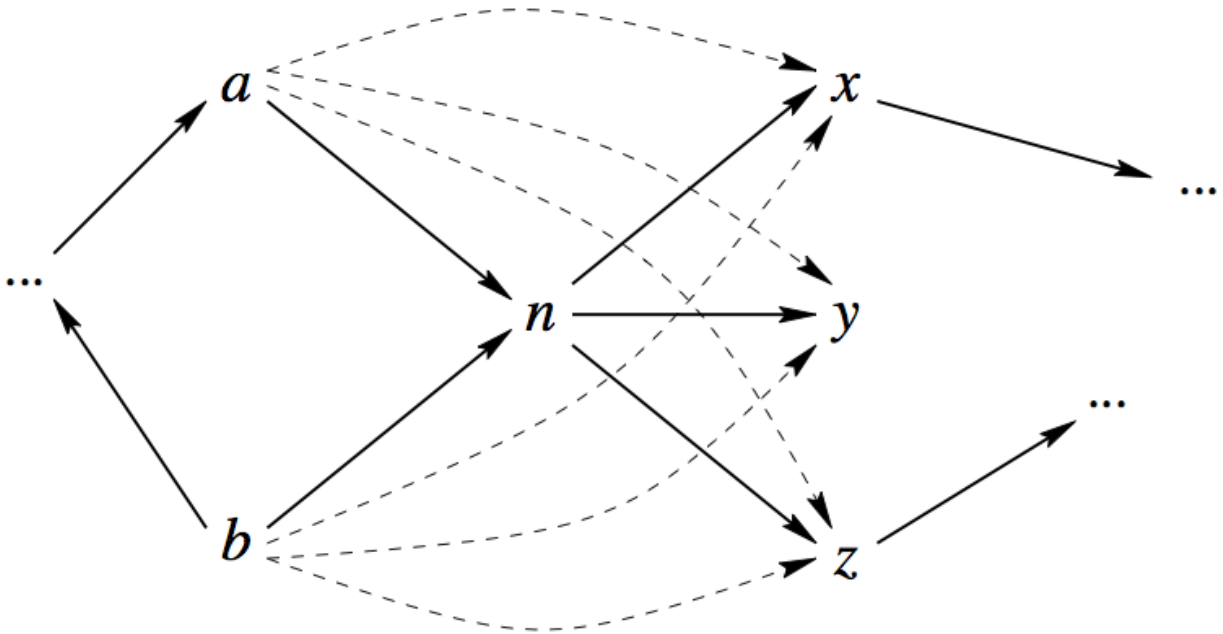
Descripción abstracta del algoritmo

La idea ...

Para cada nodo en el grafo, agregar las aristas que conecten todos los predecesores del nodo con todos sus sucesores.

El algoritmo

Para cada nodo x en el grafo G:
  Para cada nodo y en $\text{pred}(x, G)$:
    Para cada nodo z en $\text{succ}(x, G)$:
      agregue la arista (y, z) a G.



Representaciones del grafo

Para transformar el algoritmo en un programa, tenemos que escoger una representación para los grafos dirigidos.

- Lista de adyacencias. El grafo es una lista con elementos de la forma `I#Ns` donde `I` identifica un nodo y `Ns` es la lista ordenada de sus sucesores inmediatos.
- Matriz. El grafo es un arreglo de dos dimensiones. El elemento con coordenadas (`I`,`J`) es **`true`** si existe una arista del nodo `I` al nodo `J`. Si no es así, entonces ese elemento es **`false`**.

Plan

Versión declarativa

Principal

```
fun {ClauTransDecl G}
   Xs={Map G fun {$ X#_} X end}
in
   {FoldL Xs
    fun {$ InG X}
    SX={Suc X InG} in
     {Map InG
      fun {$ Y#SY}
         Y#if {Member X SY}
           then {Unión SY SX}
           else SY
           end
        end}
    end G}
end
```

Versión declarativa

Principal

```
fun {ClauTransDecl G}
   Xs={Map G fun {$ X#_} X end}
in
   {FoldL Xs
    fun {$ InG X}
    SX={Suc X InG} in
     {Map InG
      fun {$ Y#SY}
         Y#if {Member X SY}
           then {Unión SY SX}
           else SY
           end
        end}
    end G}
end
```

Auxiliares: `Suc`

```
fun {Suc X G}
  case G of
    Y#SY|G2 then
      if X==Y then
         SY
      else
         {Suc X G2}
      end
  end
end
```

Versión declarativa

Principal

```
fun {ClauTransDecl G}
   Xs={Map G fun {$ X#_} X end}
in
   {FoldL Xs
    fun {$ InG X}
    SX={Suc X InG} in
     {Map InG
      fun {$ Y#SY}
         Y#if {Member X SY}
           then {Unión SY SX}
           else SY
           end
        end}
    end G}
end
```

Auxiliares: `Unión`

```
fun {Unión A B}
 case A#B
 of nil#B then B
 [] A#nil then A
 [] (X|A2)#(Y|B2) then
    if X==Y then
       X|{Unión A2 B2}
    elseif X<Y then
       X|{Unión A2 B}
    elseif X>Y then
       Y|{Unión A B2}
    end
 end
end
```

Plan

Versión con estado

Principal

```
proc {ClauTransEstado GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for K in L..H do
  for I in L..H do
     if GM.I.K then
      for J in L..H do
         if GM.K.J then
            GM.I.J:=true
         end
      end
     end
  end
 end
end
```

Comparación de las dos soluciones

- En ambos modelos, declarativo y con estado, es razonable implementar la clausura transitiva.
- La escogencia de la representación puede ser más importante que la escogencia del modelo de computación.
- Los programas declarativos tienden a ser menos legibles que los programas con estado.

Versión con estado

Principal

```
proc {ClauTransEstado GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for K in L..H do
  for I in L..H do
     if GM.I.K then
      for J in L..H do
         if GM.K.J then
            GM.I.J:=true
         end
      end
     end
  end
 end
end
```

Comparación de las dos soluciones

- En ambos modelos, declarativo y con estado, es razonable implementar la clausura transitiva.
- La escogencia de la representación puede ser más importante que la escogencia del modelo de computación.
- Los programas declarativos tienden a ser menos legibles que los programas con estado.
- Los programas con estado tienden a ser más monolíticos que los programas declarativos.
- Es más fácil paralelizar un programa declarativo porque existen pocas dependencias entre sus partes.

Versión con estado

Principal

```
proc {ClauTransEstado GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for K in L..H do
  for I in L..H do
     if GM.I.K then
      for J in L..H do
         if GM.K.J then
            GM.I.J:=true
         end
      end
     end
  end
 end
end
```

Comparación de las dos soluciones

- En ambos modelos, declarativo y con estado, es razonable implementar la clausura transitiva.
- La escogencia de la representación puede ser más importante que la escogencia del modelo de computación.
- Los programas declarativos tienden a ser menos legibles que los programas con estado.
- Los programas con estado tienden a ser más monolíticos que los programas declarativos.
- Es más fácil paralelizar un programa declarativo porque existen pocas dependencias entre sus partes.

Versión con estado

Principal

```
proc {ClauTransEstado GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for K in L..H do
  for I in L..H do
     if GM.I.K then
      for J in L..H do
         if GM.K.J then
            GM.I.J:=true
         end
      end
     end
  end
 end
end
```

Comparación de las dos soluciones

- En ambos modelos, declarativo y con estado, es razonable implementar la clausura transitiva.
- La escogencia de la representación puede ser más importante que la escogencia del modelo de computación.
- Los programas declarativos tienden a ser menos legibles que los programas con estado.
- Los programas con estado tienden a ser más monolíticos que los programas declarativos.
- Es más fácil paralelizar un programa declarativo porque existen pocas dependencias entre sus partes.

Versión con estado

Principal

```
proc {ClauTransEstado GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for K in L..H do
  for I in L..H do
     if GM.I.K then
      for J in L..H do
         if GM.K.J then
            GM.I.J:=true
         end
      end
     end
  end
 end
end
```

Comparación de las dos soluciones

- En ambos modelos, declarativo y con estado, es razonable implementar la clausura transitiva.
- La escogencia de la representación puede ser más importante que la escogencia del modelo de computación.
- Los programas declarativos tienden a ser menos legibles que los programas con estado.
- Los programas con estado tienden a ser más monolíticos que los programas declarativos.
- Es más fácil paralelizar un programa declarativo porque existen pocas dependencias entre sus partes.

Versión con estado

Principal

```
proc {ClauTransEstado GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for K in L..H do
  for I in L..H do
     if GM.I.K then
      for J in L..H do
         if GM.K.J then
            GM.I.J:=true
         end
      end
     end
  end
 end
end
```

Comparación de las dos soluciones

- En ambos modelos, declarativo y con estado, es razonable implementar la clausura transitiva.
- La escogencia de la representación puede ser más importante que la escogencia del modelo de computación.
- Los programas declarativos tienden a ser menos legibles que los programas con estado.
- Los programas con estado tienden a ser más monolíticos que los programas declarativos.
- Es más fácil paralelizar un programa declarativo porque existen pocas dependencias entre sus partes.

Universidad del Valle

Plan

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.
- Desarrollo orientado por las pruebas.

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.
- Desarrollo orientado por las pruebas.

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.
- Desarrollo orientado por las pruebas.

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.
- Desarrollo orientado por las pruebas.

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.
- Desarrollo orientado por las pruebas.

DII

- Empezar con un conjunto pequeño de requerimientos y construir un sistema completo que los satisfaga.
- Extender continuamente los requerimientos de acuerdo a la retroalimentación del usuario, extendiendo la especificación y la arquitectura del sistema.
- Evitar la optimización prematura. La optimización del desempeño se puede realizar cerca del final del desarrollo, pero sólo si existen problemas de desempeño.
- Refactorización: Reorganizar el diseño tanto como sea necesario durante el desarrollo, para conservar una buena organización de los componentes.

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.
- Desarrollo orientado por las pruebas.

DII

- Empezar con un conjunto pequeño de requerimientos y construir un sistema completo que los satisfaga.
- Extender continuamente los requerimientos de acuerdo a la retroalimentación del usuario, extendiendo la especificación y la arquitectura del sistema.
- Evitar la optimización prematura. La optimización del desempeño se puede realizar cerca del final del desarrollo, pero sólo si existen problemas de desempeño.
- Refactorización: Reorganizar el diseño tanto como sea necesario durante el desarrollo, para conservar una buena organización de los componentes.

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.
- Desarrollo orientado por las pruebas.

DII

- Empezar con un conjunto pequeño de requerimientos y construir un sistema completo que los satisfaga.
- Extender continuamente los requerimientos de acuerdo a la retroalimentación del usuario, extendiendo la especificación y la arquitectura del sistema.
- Evitar la optimización prematura. La optimización del desempeño se puede realizar cerca del final del desarrollo, pero sólo si existen problemas de desempeño.
- Refactorización: Reorganizar el diseño tanto como sea necesario durante el desarrollo, para conservar una buena organización de los componentes.

Metodología de diseño

Administración del equipo

Asegurarse que el equipo trabaja junto, en forma coordinada. Tres principios:

- Compartimentar la responsabilidad de cada persona. Las responsabilidades deben respetar los límites entre componentes y no se deben solapar.
- El conocimiento debe ser intercambiado libremente, y no compartimentado.
- Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo.

Metodologías de diseño

- Desarrollo iterativo e incremental, DII.
- XP.
- Desarrollo orientado por las pruebas.

DII

- Empezar con un conjunto pequeño de requerimientos y construir un sistema completo que los satisfaga.
- Extender continuamente los requerimientos de acuerdo a la retroalimentación del usuario, extendiendo la especificación y la arquitectura del sistema.
- Evitar la optimización prematura. La optimización del desempeño se puede realizar cerca del final del desarrollo, pero sólo si existen problemas de desempeño.
- Refactorización: Reorganizar el diseño tanto como sea necesario durante el desarrollo, para conservar una buena organización de los componentes.

Modelos y Paradigmas de Programación

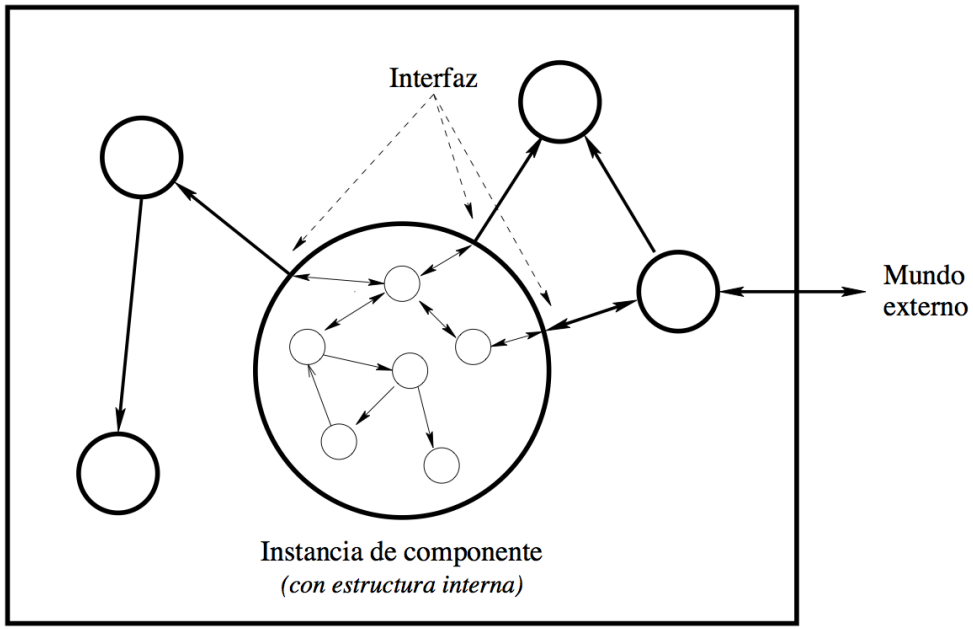
Universidad del Valle

Plan

Estructura jerárquica del sistema

¿Cómo estructurar un sistema para soportar el trabajo en equipo y el desarrollo incremental?

Grafo jerárquico con interfaces bien definidas en cada nivel



- Aplicación: conjunto de nodos interactuantes.
- Nodo: instancia de un componente; es en sí mismo una pequeña aplicación, y puede descomponerse a sí mismo en un grafo.

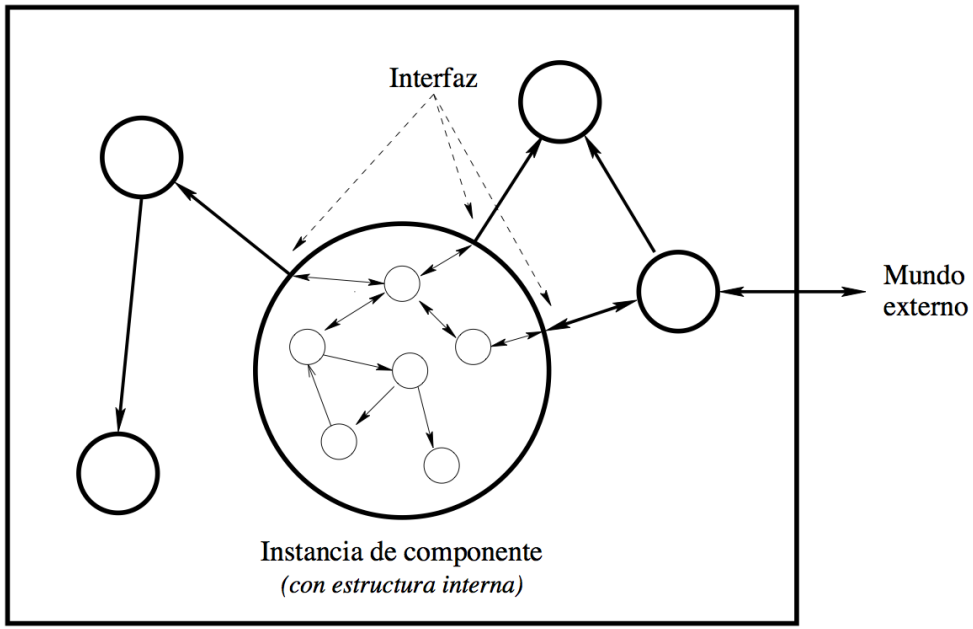
Aspectos a tener en cuenta

- Conexión entre componentes.
- Comunicación entre componentes.
- Independencia del modelo
- Compilación vs. Ejecución eficientes

Estructura jerárquica del sistema

¿Cómo estructurar un sistema para soportar el trabajo en equipo y el desarrollo incremental?

Grafo jerárquico con interfaces bien definidas en cada nivel



- Aplicación: conjunto de nodos interactuantes.
- Nodo: instancia de un componente; es en sí mismo una pequeña aplicación, y puede descomponerse a sí mismo en un grafo.

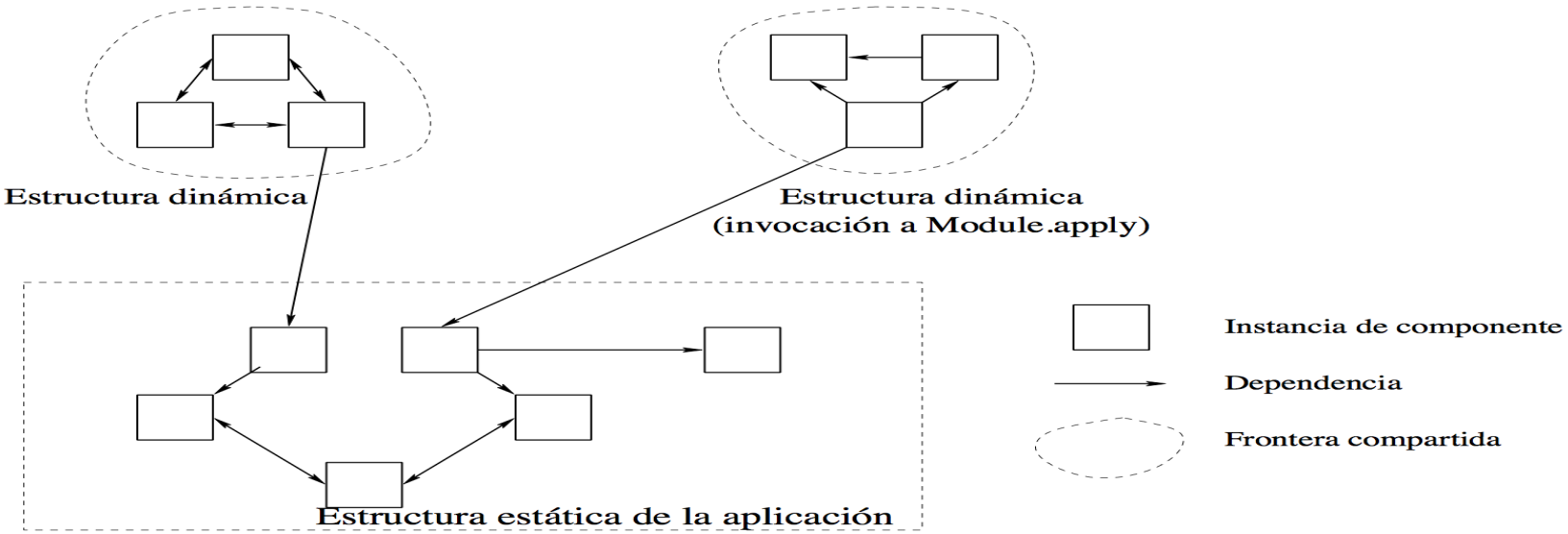
Aspectos a tener en cuenta

- Conexión entre componentes.
- Comunicación entre componentes.
- Independencia del modelo
- Compilación vs. Ejecución eficientes

Estructura jerárquica del sistema: aspectos (1)

Conexión de componentes

- Estructura estática: Grafo de componentes conocido en el momento en que se diseña la aplicación. Se pueden conectar tan pronto como la aplicación comienza su ejecución. Cada instancia corresponde a una biblioteca o un paquete. Varios componentes usan la misma instancia.
- Estructura dinámica: Creación y conexión de componentes en tiempo de ejecución. Se pueden necesitar varias instancias de un componente.



Estructura jerárquica del sistema: aspectos (2)

Comunicación de componentes: protocolos

- Procedimiento: La aplicación es secuencial y un componente invoca al otro como un procedimiento. El invocador no es necesariamente el único iniciador; pueden existir invocaciones anidadas donde el centro de control pasa hacia adelante y hacia atrás entre los componentes. Pero, existe un único centro de control global, el cual liga fuertemente los dos componentes.
- Concurrente sincrónico: cada componente evoluciona independientemente de los otros y puede iniciar y terminar comunicaciones con otros componentes, de acuerdo a algún protocolo que ambos componentes hayan acordado. Los componentes se ejecutan concurrentemente. Hay varios centros de control, denominados hilos, los cuales evolucionan independientemente. Cada componente invoca a los otros sincrónicamente, i.e., cada componente espera una respuesta para poder continuar.
- Concurrente asincrónico: Conjunto de componentes concurrentes que se comunican a través de canales asincrónicos. Cada componente envía mensajes a los otros, pero no tienen que esperar una respuesta para continuar. En esta organización, cada componente conoce la identidad del componente con el cual se comunica.

Estructura jerárquica del sistema: aspectos (2)

Comunicación de componentes: protocolos

- Procedimiento: La aplicación es secuencial y un componente invoca al otro como un procedimiento. El invocador no es necesariamente el único iniciador; pueden existir invocaciones anidadas donde el centro de control pasa hacia adelante y hacia atrás entre los componentes. Pero, existe un único centro de control global, el cual liga fuertemente los dos componentes.
- Concurrente sincrónico: cada componente evoluciona independientemente de los otros y puede iniciar y terminar comunicaciones con otros componentes, de acuerdo a algún protocolo que ambos componentes hayan acordado. Los componentes se ejecutan concurrentemente. Hay varios centros de control, denominados hilos, los cuales evolucionan independientemente. Cada componente invoca a los otros sincrónicamente, i.e., cada componente espera una respuesta para poder continuar.
- Concurrente asincrónico: Conjunto de componentes concurrentes que se comunican a través de canales asincrónicos. Cada componente envía mensajes a los otros, pero no tienen que esperar una respuesta para continuar. En esta organización, cada componente conoce la identidad del componente con el cual se comunica.

Estructura jerárquica del sistema: aspectos (2)

Comunicación de componentes: protocolos

- Procedimiento: La aplicación es secuencial y un componente invoca al otro como un procedimiento. El invocador no es necesariamente el único iniciador; pueden existir invocaciones anidadas donde el centro de control pasa hacia adelante y hacia atrás entre los componentes. Pero, existe un único centro de control global, el cual liga fuertemente los dos componentes.
- Concurrente sincrónico: cada componente evoluciona independientemente de los otros y puede iniciar y terminar comunicaciones con otros componentes, de acuerdo a algún protocolo que ambos componentes hayan acordado. Los componentes se ejecutan concurrentemente. Hay varios centros de control, denominados hilos, los cuales evolucionan independientemente. Cada componente invoca a los otros sincrónicamente, i.e., cada componente espera una respuesta para poder continuar.
- Concurrente asincrónico: Conjunto de componentes concurrentes que se comunican a través de canales asincrónicos. Cada componente envía mensajes a los otros, pero no tienen que esperar una respuesta para continuar. En esta organización, cada componente conoce la identidad del componente con el cual se comunica.

Estructura jerárquica del sistema: aspectos (3)

Principio de independencia del modelo

La interfaz de un componente debe ser independiente del modelo de computación utilizado para implementar el componente. La interfaz solamente debe depender de la funcionalidad externamente visible del componente.

Estructura jerárquica del sistema: aspectos (3)

Principio de independencia del modelo

La interfaz de un componente debe ser independiente del modelo de computación utilizado para implementar el componente. La interfaz solamente debe depender de la funcionalidad externamente visible del componente.

Compilación vs. ejecución eficientes

Dos deseos:

- Compilar un componente tan rápida y eficientemente como sea posible: compilar de manera separada un componente, i.e., sin saber nada sobre los otros componentes.
- Que el programa final, con todos los componentes ensamblados, sea tan eficiente y compacto como sea posible: requiere análisis en tiempo de compilación (comprobación de tipos, inferencia de tipos, u optimización global).
- En Mozart los componentes se pueden compilar sin saber nada sobre los otros. La compilación es completamente escalable, pero hay un menor grado de optimización y los errores de tipo sólo se detectan en tiempo de ejecución.

Estructura jerárquica del sistema: aspectos (3)

Principio de independencia del modelo

La interfaz de un componente debe ser independiente del modelo de computación utilizado para implementar el componente. La interfaz solamente debe depender de la funcionalidad externamente visible del componente.

Compilación vs. ejecución eficientes

Dos deseos:

- Compilar un componente tan rápida y eficientemente como sea posible: compilar de manera separada un componente, i.e., sin saber nada sobre los otros componentes.
- Que el programa final, con todos los componentes ensamblados, sea tan eficiente y compacto como sea posible: requiere análisis en tiempo de compilación (comprobación de tipos, inferencia de tipos, u optimización global).
- En Mozart los componentes se pueden compilar sin saber nada sobre los otros. La compilación es completamente escalable, pero hay un menor grado de optimización y los errores de tipo sólo se detectan en tiempo de ejecución.

Estructura jerárquica del sistema: aspectos (3)

Principio de independencia del modelo

La interfaz de un componente debe ser independiente del modelo de computación utilizado para implementar el componente. La interfaz solamente debe depender de la funcionalidad externamente visible del componente.

Compilación vs. ejecución eficientes

Dos deseos:

- Compilar un componente tan rápida y eficientemente como sea posible: compilar de manera separada un componente, i.e., sin saber nada sobre los otros componentes.
- Que el programa final, con todos los componentes ensamblados, sea tan eficiente y compacto como sea posible: requiere análisis en tiempo de compilación (comprobación de tipos, inferencia de tipos, u optimización global).
- En Mozart los componentes se pueden compilar sin saber nada sobre los otros. La compilación es completamente escalable, pero hay un menor grado de optimización y los errores de tipo sólo se detectan en tiempo de ejecución.

Plan

Mantenimiento

¿Cuál es la mejor forma de estructurar los programas de manera que se puedan mantener con facilidad?

Principios en el diseño de componentes

- Encapsular decisiones de diseño: cuando la decisión de diseño se modifica, solamente ese componente tiene que ser modificado.
- Evitar modificar interfaces de componentes: las interfaces de los componentes más frecuentemente necesitados deben ser diseñadas tan cuidadosamente como sea posible desde el comienzo.

Principios en el diseño de sistemas

- El menor número posible de dependencias externas.
- El menor número posible de niveles de referencias indirectas.
- Las dependencias deben ser predecibles.
- Tomar decisiones en el nivel correcto.
- Violaciones a los principios documentadas.
- Jerarquía de empaquetamiento sencilla.

Mantenimiento

¿Cuál es la mejor forma de estructurar los programas de manera que se puedan mantener con facilidad?

Principios en el diseño de componentes

- Encapsular decisiones de diseño: cuando la decisión de diseño se modifica, solamente ese componente tiene que ser modificado.
- Evitar modificar interfaces de componentes: las interfaces de los componentes más frecuentemente necesitados deben ser diseñadas tan cuidadosamente como sea posible desde el comienzo.

Principios en el diseño de sistemas

- El menor número posible de dependencias externas.
- El menor número posible de niveles de referencias indirectas.
- Las dependencias deben ser predecibles.
- Tomar decisiones en el nivel correcto.
- Violaciones a los principios documentadas.
- Jerarquía de empaquetamiento sencilla.

Mantenimiento

¿Cuál es la mejor forma de estructurar los programas de manera que se puedan mantener con facilidad?

Principios en el diseño de componentes

- Encapsular decisiones de diseño: cuando la decisión de diseño se modifica, solamente ese componente tiene que ser modificado.
- Evitar modificar interfaces de componentes: las interfaces de los componentes más frecuentemente necesitados deben ser diseñadas tan cuidadosamente como sea posible desde el comienzo.

Principios en el diseño de sistemas

- El menor número posible de dependencias externas.
- El menor número posible de niveles de referencias indirectas.
- Las dependencias deben ser predecibles.
- Tomar decisiones en el nivel correcto.
- Violaciones a los principios documentadas.
- Jerarquía de empaquetamiento sencilla.