Escuela Superior de Ingeniería y Tecnología
Universidad de La Laguna

Grado en Ingeniería Informática
Algoritmos y Estructuras de Datos Avanzadas

Curso 2021-2022

Práctica 2: El ciclo de la vida

1. Objetivo

En esta práctica se trabaja el manejo de objetos en memoria dinámica y la utilización del polimorfismo dinámico para implementar distintos comportamientos de un objeto en función de su estado (**Patrón de diseño State** [5]).

2. Entrega

Se realizará en una sesión de entrega en el laboratorio entre el 14 y el 18 de marzo de 2022.

3. Enunciado

En la práctica anterior [1] se implementó una versión del Juego de la Vida [3] de Conway. Se trata de un autómata celular [2] donde cada célula se encuentra en uno de dos posibles estados (“viva” o “muerta”). Las células se extienden sobre una rejilla bidimensional y para cada célula se utiliza la vecindad de Moore (sus ocho células contiguas) para definir unas reglas de transición que toman en cuenta tanto el estado de la célula como el estado de las células de su vecindad.

En esta práctica se pretende generalizar el código de la práctica anterior para permitir representar comportamientos más complejos. Para ello se extiende el conjunto de estados en los que puede encontrarse cada célula. Por ejemplo, usando los estados del ciclo de vida de una mariposa monarca [4]: “huevo”, “larva”, “pupa” y “adulta”. Además, se mantiene el estado “muerta” para representar la situación de ausencia del resto de estados.

Para cada uno de los estados es necesario establecer una función de transición que defina los cambios posibles a otros estados teniendo en cuenta los estados de las células vecinas. Para no complicar la implementación vamos a asumir las siguientes reglas en las que sólo se tiene en cuenta la cantidad de los diferentes estados en las células de la vecindad.

- Una célula en estado “muerta” transita a estado “huevo” cuando dentro de su vecindad existen al menos dos células en estado “adulta”. En cualquier otro caso permanece en estado “muerta”.
- Una célula en estado “huevo” transita a estado “muerta” cuando dentro de su vecindad el número de células en estado “larva” es mayor que el número de células en estado “huevo”. En cualquier otro caso transita a estado “larva”.
- Una célula en estado “larva” transita a estado “muerta” cuando dentro de su vecindad el número de células en estado “larva” es mayor que la suma de células en los estados “huevo”, “pupa” y adulta. En cualquier otro caso transita a estado “pupa”.
- Una célula en estado “pupa” transita a estado “muerta” cuando dentro de su vecindad el número de células en estado “larva” es mayor que el número de células en cualquier otro estado. En cualquier otro caso transita a estado “adulta”.
- Una célula en estado “adulta” transita a estado “huevo” cuando dentro de su vecindad hay al menos otra célula en estado “adulta”. En cualquier otro caso transita a estado “muerta”.

Las reglas de transición podrán modificarse para alcanzar la continuidad y estabilidad del sistema.

4. Notas de implementación

En cada turno de evolución una célula puede encontrarse en un único estado entre un número finito de estados, y el comportamiento de la célula (sus reglas de transición) puede ser diferente en cada estado. Para implementar esta situación se utilizará el **Patrón de Diseño State** [5].

1. La implementación del estado de una célula se realizará mediante la clase abstracta `State`. En esta clase virtual sólo se definen los métodos nulos que son comunes a cualquiera de los estados de una célula. Al ser métodos nulos la implementación del comportamiento se realizará en las clases derivadas que representan a cada estado concreto. Los comportamientos que pueden variar según el estado son:

    a. La forma de definir la vecindad, así como la cuenta de los tipos de las células vecinas de la que dependen las reglas de transición, se realiza en el método:

    ```
    virtual int State::neighbors(const Grid&, int i, int j)=0;
    ```

    b. La implementación de las reglas de transición para calcular el siguiente estado:

    ```
    virtual State* State::nextState()=0;
    ```

    c. La forma de identificar cada estado. Utilizaremos la primera letra del nombre de cada estado, un carácter del código ASCII, para visualizar los distintos estados de una célula; salvo el estado “muerta” que se visualizará con el carácter `' '`.

    ```
    virtual char State::getState() const =0;
    ```

2. A partir de la clase base `State` se derivan las clases `StateEgg`, `StateLarva`, `StatePupa`, `StateAdult` y `StateDead` que representan los **estados concretos** del **patrón State** e implementan las operaciones polimórficas.

3. El objeto célula, `Cell`, se corresponde con la clase **contexto** del **patrón State**.

    a. El estado de la célula es un atributo privado que se implementa mediante un puntero a la clase abstracta `State`. Por tanto se requieren los métodos de acceso. El método getter retorna el carácter ASCII que identifica al estado. Y el método setter actualiza el objeto estado al que apunta el atributo.

    ```
    char Cell::getState() const;
    void Cell::setState(State*);
    ```

    b. La posición `(i,j)` que ocupa una célula en la rejilla también es un atributo privado.

    c. Cada célula es responsable de actualizar su estado, cuando le corresponda en el turno actual. Este método delega su responsabilidad en los métodos del objeto `State` que aplica la regla de transición que corresponde y actualiza el objeto `State` si es necesario.

    ```
    void Cell::updateState();
    ```

    d. La interacción con las células de la vecindad también se delega en el objeto `State` contenido en la célula.

    ```
    int Cell::neighbors(const Grid&);
    ```

e. La sobrecarga del operador de inserción en flujo utiliza el carácter que representa al objeto estado contenido en la célula.

```
ostream& operator<<(ostream&, const Cell&);
```

4. El objeto rejilla, `Grid`, contiene un array de `NxM` células. Se corresponde con el objeto **cliente** del **patrón State**. Este objeto es responsable de crear y almacenar las células que constituyen el juego y de establecer su valor de estado inicial.

   a. El constructor recibe el tamaño de la rejilla y crea el array de células en memoria dinámica. Cada célula se inicia con la posición `(i,j)` que ocupa en la rejilla y el valor de su estado en el inicio. Todas las células están inicialmente en estado "muerta", salvo aquellas que el usuario indique en tiempo de ejecución que deben estar en otro estado en el turno inicial.

   b. La rejilla da acceso de sólo lectura a cualquiera de sus células.

```
const Cell& Grid::getCell(int, int) const;
```

   c. El objeto rejilla es responsable de contar las unidades de tiempo, **turnos**, de forma que en cada turno todas las células actualizan su estado a partir de los valores del estado de las células en el turno anterior. Esta operación se implementa en el método:

```
void Grid::nextGeneration();
```

   d. Para garantizar que la actualización de las células en cada turno tiene en cuenta los valores del turno anterior, la rejilla se recorre dos veces. En el primer recorrido cada célula cuenta sus vecinas; y en el segundo recorrido cada célula actualiza su estado.

   e. En cada turno se realiza un tercer recorrido de la rejilla para que cada célula se muestra en pantalla de la siguiente forma:

Turno 0:

```
HH  PA
LA  A P LHA
      L
```

   f. Hay que tener en cuenta que las células del borde de la rejilla no cumplen la condición de tener 8 células vecinas. Para evitar la complicación en el cómputo de las condiciones de frontera se creará una rejilla con las dimensiones `(N+2)x(M+2)`, pero sólo se actualizan y visualizan las `NxM` células del interior de la rejilla, que son las que cumplen la condición de tener las 8 células vecinas.

5. El funcionamiento del programa principal es el siguiente:

   a. Solicita el tamaño de la rejilla, N y M, y el número máximo de turnos que tendrá el juego.

b. Crea un objeto `Grid` con todas sus células en estado “muerta”.

c. Solicita las posiciones de las células y el estado que deben tener en en el turno 0 y actualiza el estado en dichas células de la rejilla.

d. Muestra por pantalla el estado de la rejilla.

e. Ejecuta un bucle donde cada iteración se corresponde a un turno en la evolución de juego. En cada turno se actualizan y se muestran por pantalla el estado de las células en la rejilla.

6. Durante las sesiones de laboratorio se podrán solicitar cambios y/o ampliaciones en la especificación de este enunciado.

5. Referencias

[1] Moodle: Enunciado de la Práctica 1.
[2] Wikipedia: Autómata celular. https://es.m.wikipedia.org/wiki/Autómata_celular
[3] Wikipedia: Juego de la vida. https://es.m.wikipedia.org/wiki/Juego_de_la_vida
[4] Arizona State University: Ciclo de vida de la mariposa monarca. https://askabiologist.asu.edu/content/la-vida-monarca
[5] Refactoring Guru: Patrón de Diseño State. https://refactoring.guru/es/design-patterns/state