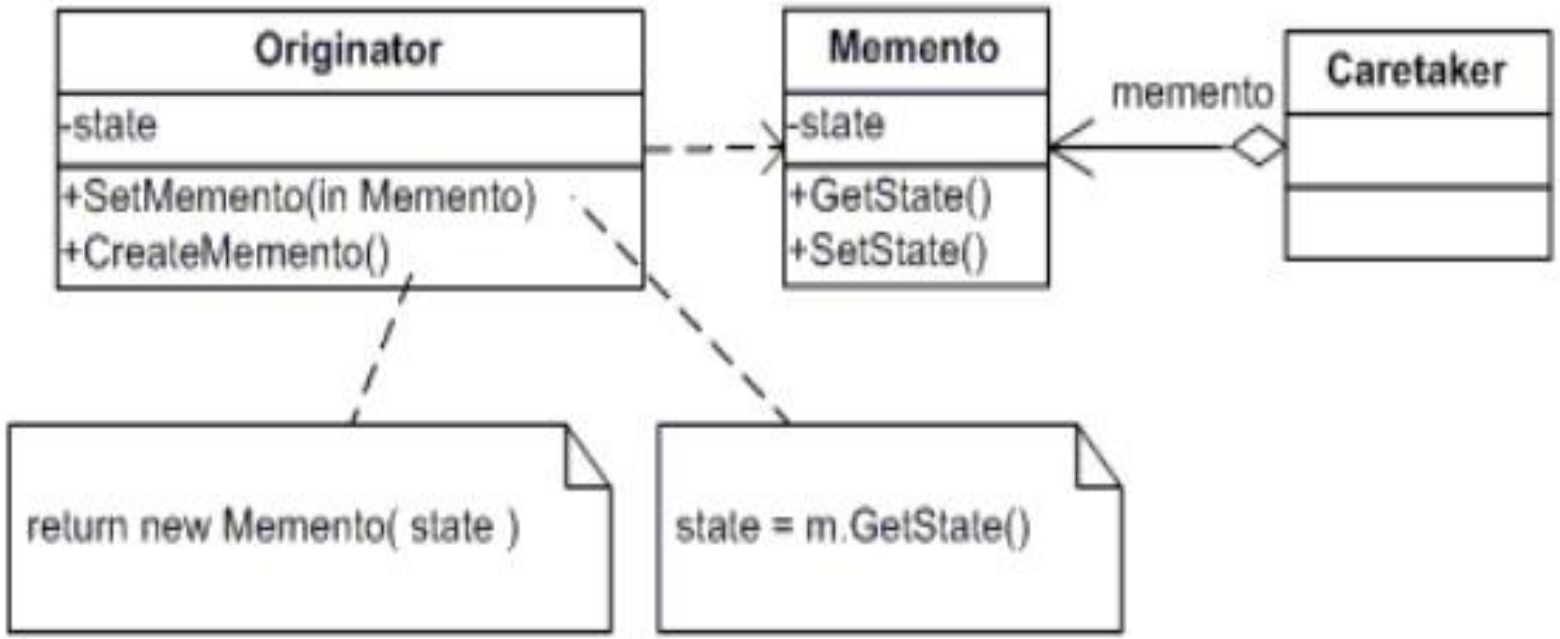
Memento Code

JIM FAWCETT
CSE776 – DESIGN PATTERNS
FALL 2016

UML class diagram

www.dofactory.com



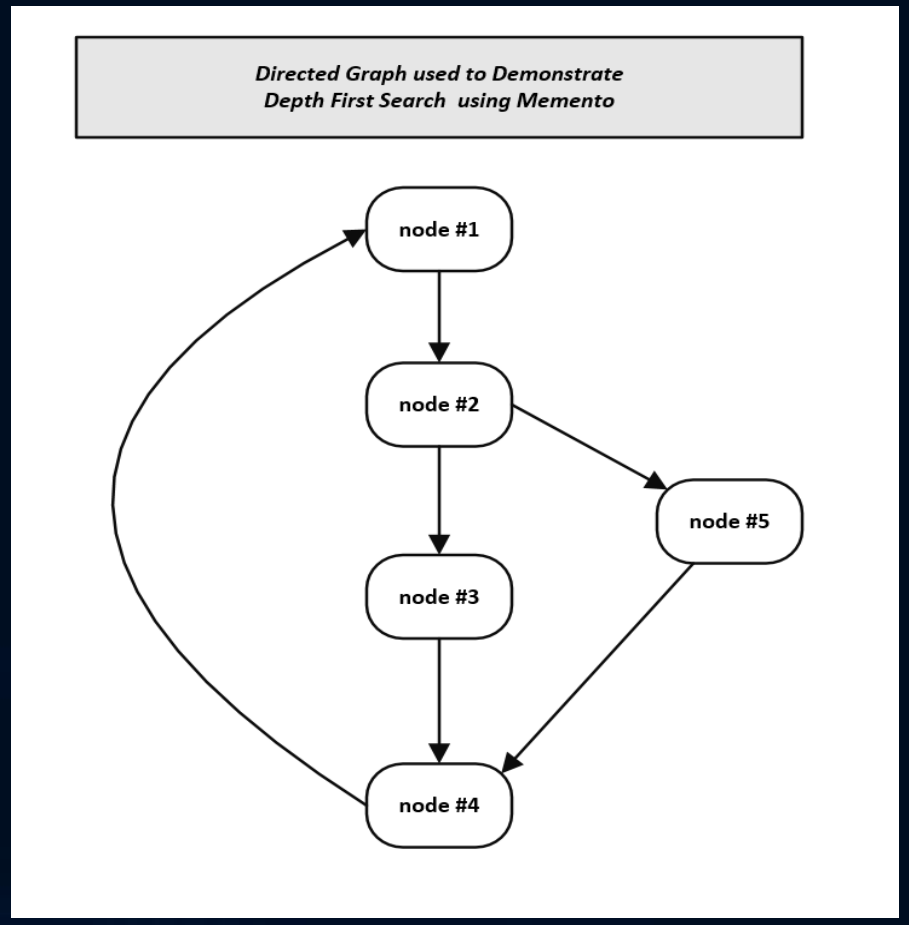
Memento Pattern:
Non-Recursive Depth First Search

```
 Demonstrating Depth First Search using Memento Pattern
=======================================================

 starting at node 1:
---------------------
 visiting node #1
 visiting node #2
 visiting node #3
 visiting node #4
 visiting node #3
 visiting node #2
 visiting node #5
 visiting node #2


 restarting at node 3:
-----------------------
 visiting node #3
 visiting node #4
 visiting node #1
 visiting node #2
 visiting node #5
 visiting node #2
 visiting node #1
 visiting node #4

Press any key to continue . . .
```



Node – used to build graph

```
template<class NodeVal> class Node {

  public:
    Node(NodeVal v);
    NodeVal& value();
    bool& marked();
    Node<NodeVal>* getUnmarkedChild();
    void attachChild(Node<NodeVal>* pNode);

  private:
    std::vector< Node<NodeVal>* > _children;
    NodeVal _value;
    bool _visited;
};
//----< initialize node with value and marking >----------------------

template<class NodeVal>
Node<NodeVal>::Node(NodeVal v) : _value(v), _visited(false) { }

//----< return value type >-------------------------------------------

template<class NodeVal>
NodeVal& Node<NodeVal>::value() { return _value; }
```

```
//----< get or set node marking >-------------------------------------

template<class NodeVal>
bool& Node<NodeVal>::marked() { return _visited; }

//----< return an unmarked child pointer if any exist >---------------

template<class NodeVal>
Node<NodeVal>* Node<NodeVal>::getUnmarkedChild() {

  vector< Node<NodeVal>* >::iterator it;
  for(it=_children.begin(); it!=_children.end(); it++) {
    if((*it)->marked() == false) {
      return (*it);
    }
  }
  return NULL;
}
//----< attach a child node to this node >----------------------------

template<class NodeVal>
void Node<NodeVal>::attachChild(Node<NodeVal>* pNode) {

  _children.push_back(pNode);
}
```

Originator – Node Walker

```
template<class NodeVal> class Walker {

  friend class Memento<NodeVal>;

  public:
    Walker(Node<NodeVal> *start);
    void walk();
    void SetMemento(Memento<NodeVal> *m);
    Memento<NodeVal>* CreateMemento();

  private:
    Node<NodeVal> *_currPos;
    Caretaker<NodeVal> _ct;
};
//----< initialize Walker with node, register with CareTaker >---------

template<class NodeVal>
Walker<NodeVal>::Walker(Node<NodeVal> *start) : _currPos(start) {

  _ct.acceptWalker(this);
}
```

```
//----< pass current node to memento >---------------------------------

template<class NodeVal>
void Walker<NodeVal>::SetMemento(Memento<NodeVal> *m) { m->GetState(); }

//----< create a memento and initialize its state >--------------------

template<class NodeVal>
Memento<NodeVal>* Walker<NodeVal>::CreateMemento() {

  Memento<NodeVal> *pMem = new Memento<NodeVal>(this);
  pMem->GetState();
  return pMem;
}
//----< non-recursive graph walk >-------------------------------------

template<class NodeVal>
void Walker<NodeVal>::walk() {

  Node<NodeVal>* pNode;
  do
  {
    _currPos->marked() = true;
    cout << "\n  visiting node #" << (_currPos->value());
    while((pNode = _currPos->getUnmarkedChild()) != 0)
    {
      _ct.acceptMemento();
      _currPos = pNode;
      _currPos->marked() = true;
      cout << "\n  visiting node #" << (_currPos->value());
    }
    _ct.returnMemento();
  } while(_ct.numMementos() > 0);
}
```

```
////////////////////////////////////////////////////////////////////////
// class Memento<NodeVal>

template<class NodeVal> class Memento {

  public:
    Memento(Walker<NodeVal> *pWalker);
    void GetState();
    void SetState();

  private:
    Walker<NodeVal> *_pWalker;
    Node<NodeVal> *_pNode;
};
//----< initialize Memento with Walker reference >---------------------

template<class NodeVal>
Memento<NodeVal>::Memento(Walker<NodeVal> *pWalker) : _pWalker(pWalker) { }

//----< Get Walker state and store it >--------------------------------

template<class NodeVal>
void Memento<NodeVal>::GetState() { _pNode = _pWalker->_currPos; }

//----< Set Walker state from Memento store >--------------------------

template<class NodeVal>
void Memento<NodeVal>::SetState() { _pWalker->_currPos = _pNode; }
```

Memento

- takes reference to originator
- graph walker is originator
- saves and returns location

Caretaker

```
template<class NodeVal> class Caretaker {

  public:
    void acceptWalker(Walker<NodeVal> *pWalk);
    void acceptMemento();
    int  returnMemento();
    int  numMementos() { return _mementos.size(); }

  private:
    Walker<NodeVal> *_pWalk;
    std::vector< Memento<NodeVal>* > _mementos;
};
//----< register walker with Caretaker >-----------------------------

template<class NodeVal>
void Caretaker<NodeVal>::acceptWalker(Walker<NodeVal> *pWalk)
{
  _pWalk = pWalk;
}
```

```
//----< pass memento to Caretaker for safekeeping >--------------------

template<class NodeVal>
void Caretaker<NodeVal>::acceptMemento() {

  Memento<NodeVal> *pMem = _pWalk->CreateMemento();
  _mementos.push_back(pMem);
}
//----< restore earlier state in Walker, destroy memento >-------------

template<class NodeVal>
int Caretaker<NodeVal>::returnMemento() {

  if(_mementos.size() > 0) {
    Memento<NodeVal> *pMem = _mementos.back();
    _mementos.pop_back();
    pMem->SetState();
    delete pMem;
  }
  return _mementos.size();
}
```

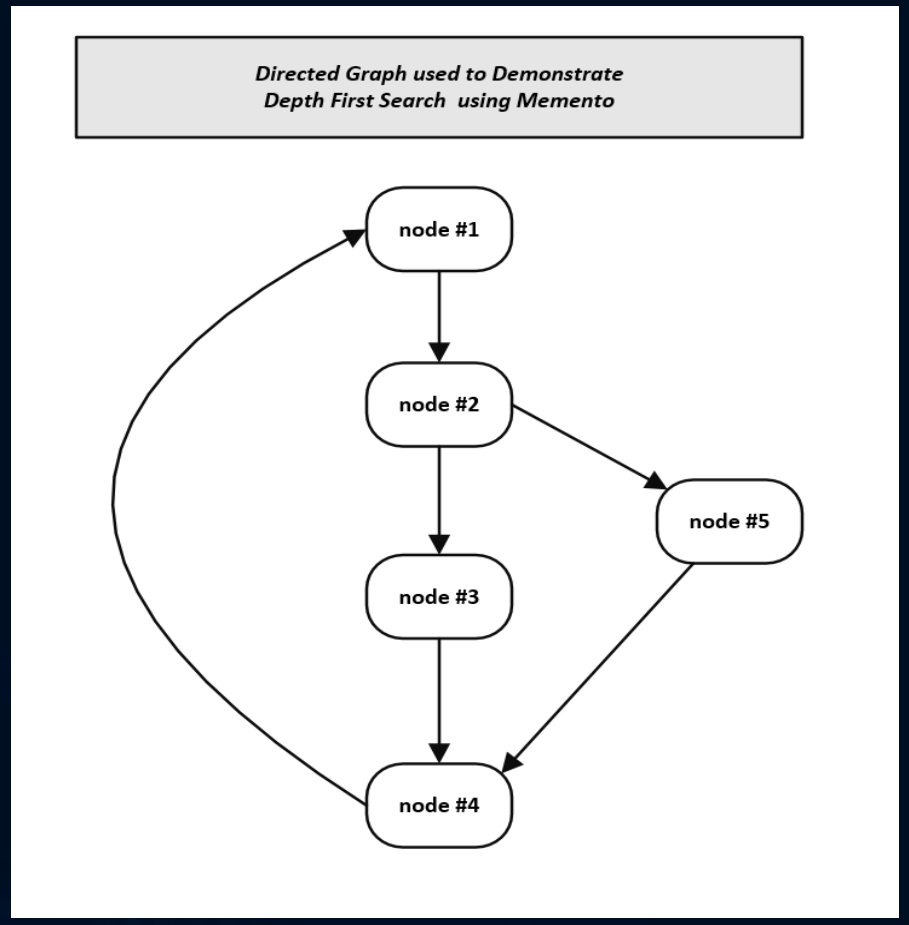
Memento Pattern:
Non-Recursive Depth First Search

```
 Demonstrating Depth First Search using Memento Pattern
=======================================================

 starting at node 1:
---------------------
 visiting node #1
 visiting node #2
 visiting node #3
 visiting node #4
 visiting node #3
 visiting node #2
 visiting node #5
 visiting node #2


 restarting at node 3:
-----------------------
 visiting node #3
 visiting node #4
 visiting node #1
 visiting node #2
 visiting node #5
 visiting node #2
 visiting node #1
 visiting node #4

Press any key to continue . . .
```



That’s All Folks!

ARRIVEDERCHI