Memento Pattern Code

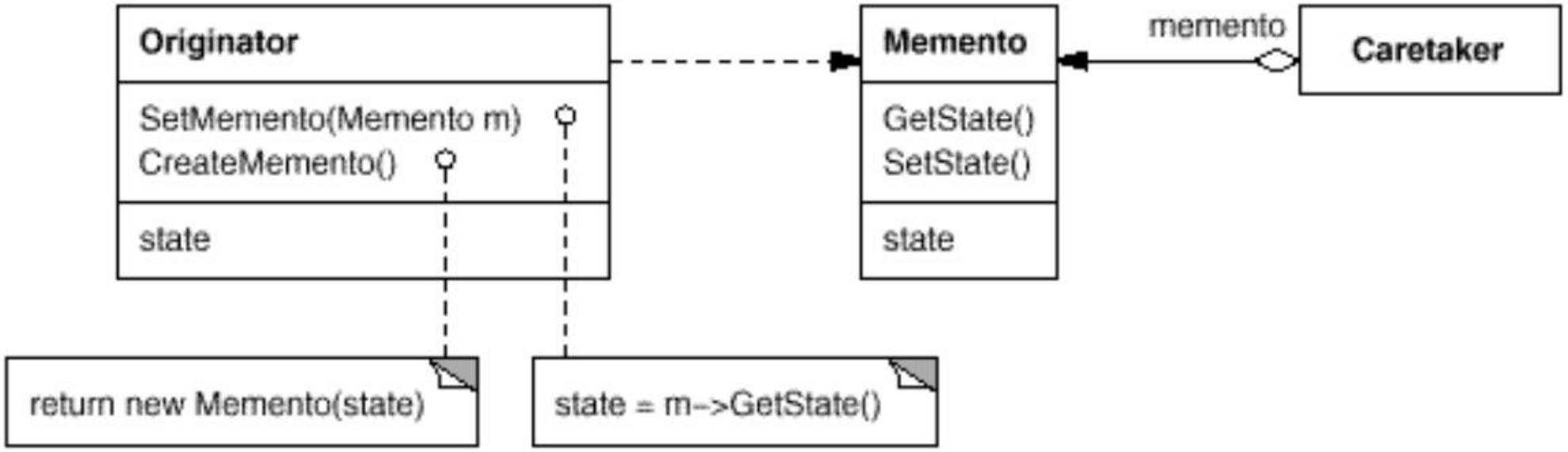
Jim Fawcett

CSE776 – Design Patterns

Fall 2018

Memento Classes

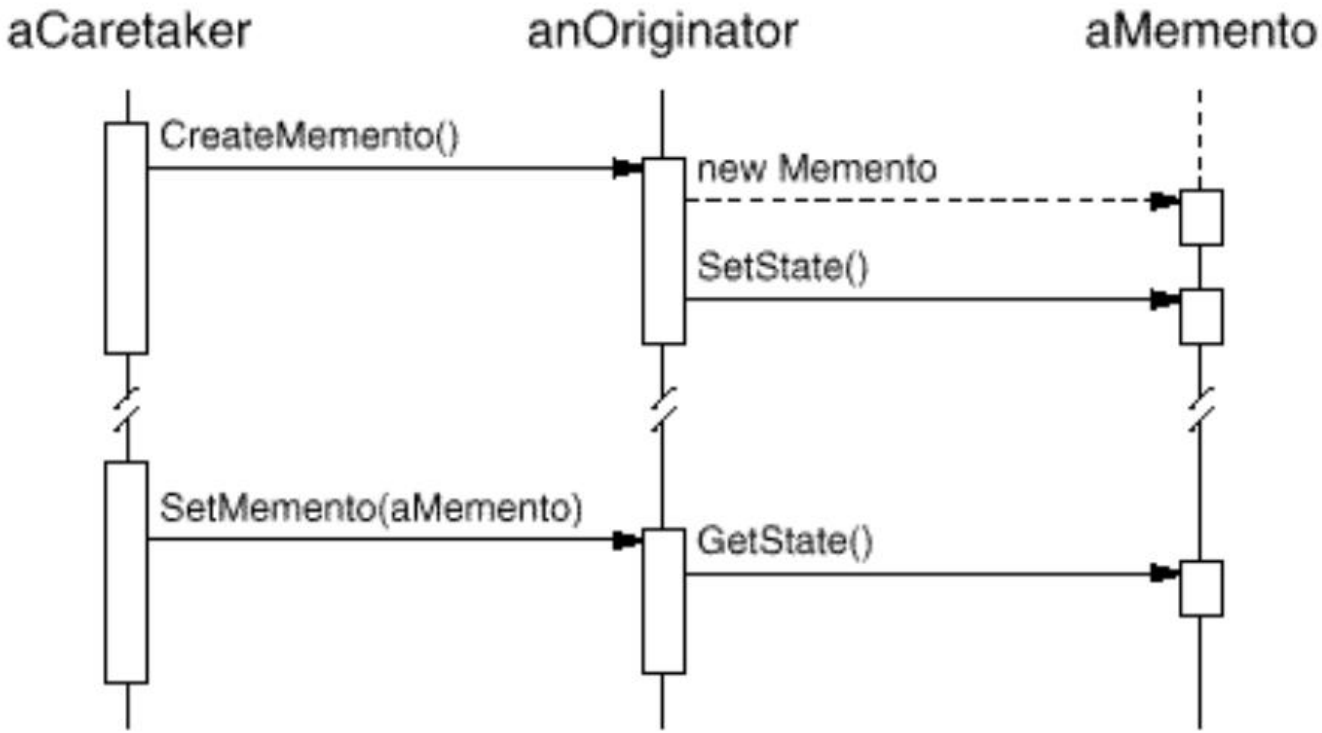
- Originator creates memento containing a snapshot of its current state
  - Uses memento to restore its internal state
- Caretaker stores memento(s) without examining or altering its state



https://paginas.fe.up.pt/~aaguiar/as/gof/hires/pat5ffso.htm

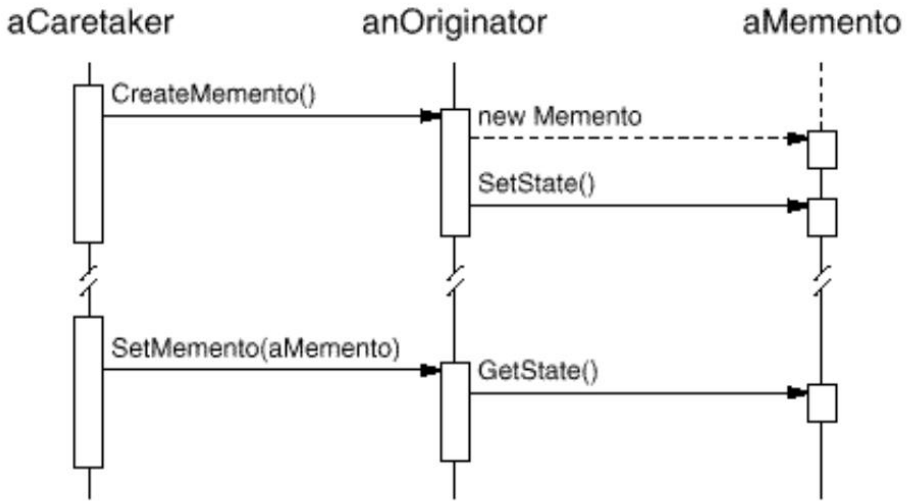
Memento Sequence Diagram

- Caretaker requests memento from Originator
- Holds it for a while
- Passes it back to the Originator



https://paginas.fe.up.pt/~aaguiar/as/gof/hires/pat5ffso.htm

Memento Class

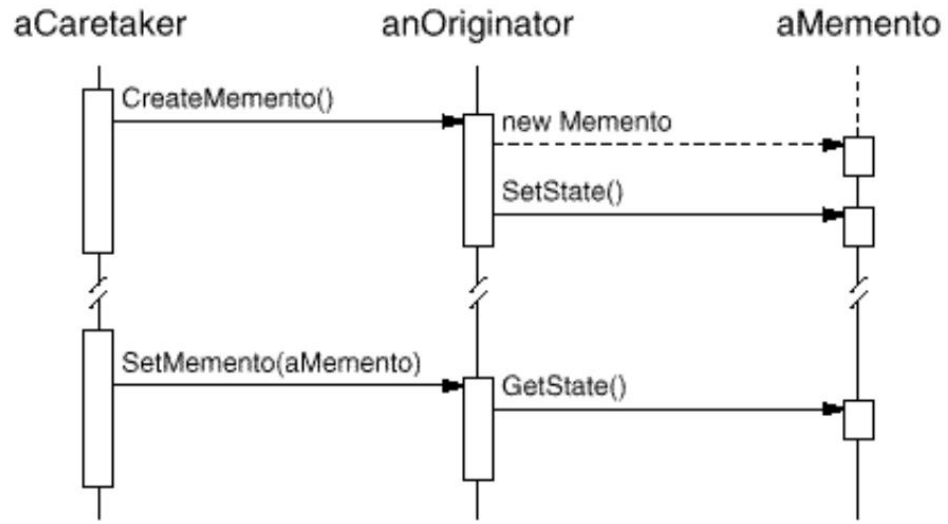



```
template<class state> class Originator;

template<class state> class Memento {

  public:
    Memento(Originator<state> *pOrig);
    void GetState();
    void SetState();

  private:
    Originator<state>* _pOrig;
    state _state;
};
```

```
template<class state>
Memento<state>::Memento(Originator<state> *pOrig) :
  _pOrig(pOrig) { }
```

```
template<class state>
void Memento<state>::GetState() {
  _state = _pOrig->_state;
}
```

```
template<class state>
void Memento<state>::SetState() {
  _pOrig->_state = _state;
}
```



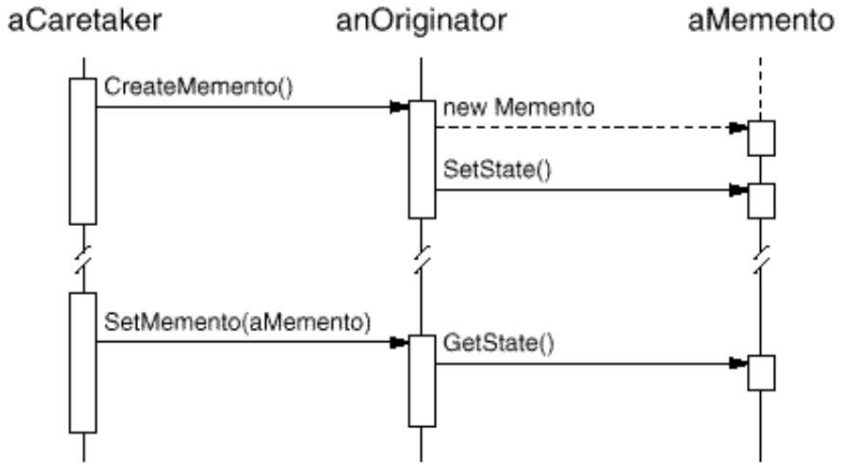
Originator Class

```
template<class state> class Originator {
  friend class Memento<state>;
  public:
    Originator(const state &s);
    void Operation(std::string s);
    void SetMemento(Memento<state> *m);
    Memento<state>* CreateMemento();
  private:
    state _state;
};

template<class state>
Originator<state>::Originator(const state &s) :
 _state(s) { }
```

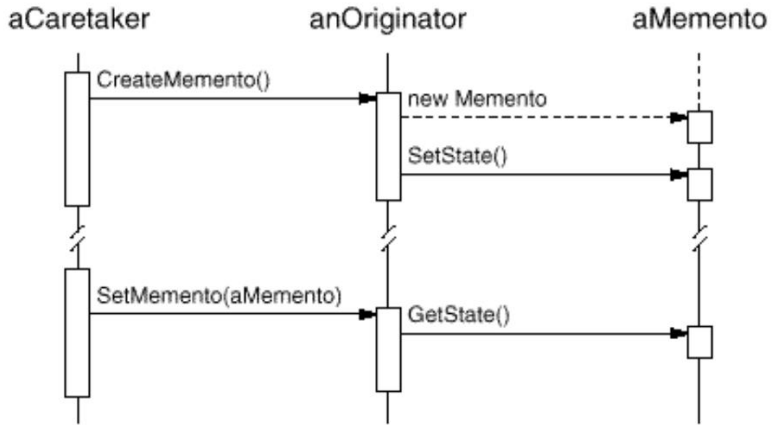
```
template<class state>
void Originator<state>::SetMemento(Memento<state> *m)
{
  m->SetState();
  cout << "\n\n  Originator retrieving state \"" << _state << "\"";
}

template<class state>
Memento<state>* Originator<state>::CreateMemento() {
  cout << "\n  Originator creating memento with state \""
       << _state << "\"";
  Memento<state> *pMem = new Memento<state>(this);
  pMem->GetState();
  return pMem;
}

template<class state>
void Originator<state>::Operation(std::string s) {
  cout << "\n\n  Originator changing state to \""
       << s << "\"";
  _state = s;
}
```

Caretaker Class

```
template<class state> class Caretaker {

  public:
    Caretaker(Originator<state> *orig);
    void acceptMemento();
    void returnMemento();

  private:
    Originator<state>* _pOrig;
    std::vector< Memento<state>* > _mementos;
};

template<class state>
Caretaker<state>::Caretaker(Originator<state> *pOrig) :
   _pOrig(pOrig) { }
```




```
template<class state>
void Caretaker<state>::acceptMemento() {
  Memento<state> *pMem = _pOrig->CreateMemento();
  _mementos.push_back(pMem);
}

template<class state>
void Caretaker<state>::returnMemento() {
  if(_mementos.size() > 0) {
    Memento<state> *pMem = _mementos.back();
    _mementos.pop_back();
    _pOrig->SetMemento(pMem);
    delete pMem;
  }
}
```

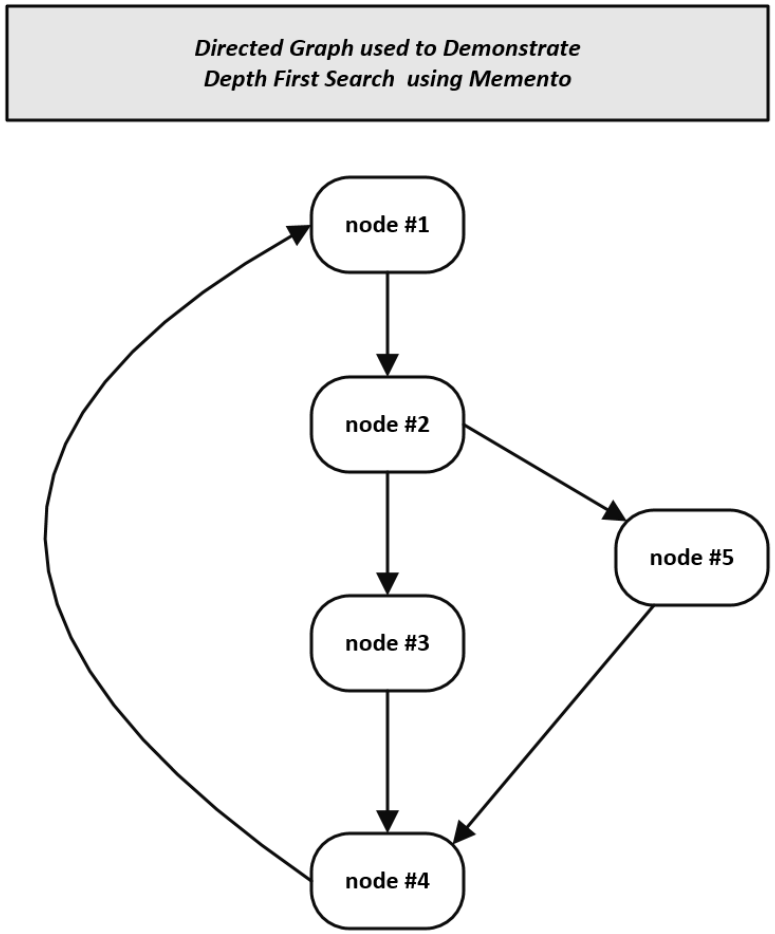



```
class State {

  public:
    State() : _str("") { }
    State(const string &s) : _str(s) { }
    string str() { return _str; }
  private:
    std::string _str;
};

ostream& operator<<(ostream &out, State s) {

  out << s.str();
  return out;
}
```

Client

```
void main() {

  cout << "\n  Demonstrating Memento Skeleton Code "
       << "\n ====================================";

  Originator<State> orig(State("first state"));
  Caretaker<State> ct(&orig);

  ct.acceptMemento();

  orig.Operation("second state");
  ct.acceptMemento();

  orig.Operation("third state");
  ct.acceptMemento();

  ct.returnMemento();
  ct.returnMemento();
  ct.returnMemento();

  cout << "\n\n";
}
```

Using Memento to do non-recursive graph walk




```
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
```

Using Memento to do non-recursive graph walk

```
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;
};

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

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

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

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;
}

template<class NodeVal>
void Node<NodeVal>::attachChild(Node<NodeVal>* pNode) {
  _children.push_back(pNode);
}
```

Graph walk Caretaker

```
template<class NodeVal> class Memento;

template<class NodeVal> class Walker;  // originator

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;
};
```

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

template<class NodeVal>
void Caretaker<NodeVal>::acceptMemento() {
  Memento<NodeVal> *pMem = _pWalk->CreateMemento();
  _mementos.push_back(pMem);
}

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();
}
```

Graph walk Originator

```
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;
};

template<class NodeVal>
Walker<NodeVal>::Walker(Node<NodeVal> *start) :
   _currPos(start) {
  _ct.acceptWalker(this);
}
```

```
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);
}

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

template<class NodeVal>
Memento<NodeVal>* Walker<NodeVal>::CreateMemento() {
  Memento<NodeVal> *pMem = new Memento<NodeVal>(this);
  pMem->GetState();
  return pMem;
}
```

Graph walk Memento

```
template<class NodeVal> class Memento {
  public:
    Memento(Walker<NodeVal> *pWalker);
    void GetState();
    void SetState();
  private:
    Walker<NodeVal> *_pWalker;
    Node<NodeVal> *_pNode;
};

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

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

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

```
Node<int> n1(1), n2(2), n3(3), n4(4), n5(5);

n1.attachChild(&n2);
n2.attachChild(&n3); n2.attachChild(&n5);
n3.attachChild(&n4);
n4.attachChild(&n1);
n5.attachChild(&n4);

cout << "\n  starting at node 1: "
     << "\n ---------------------";
Walker<int> walker(&n1);
walker.walk();
cout << "\n\n";
cout << "\n  restarting at node 3: "
     << "\n ----------------------";
n1.marked() = false;
n2.marked() = false;
n3.marked() = false;
n4.marked() = false;
n5.marked() = false;

Walker<int> walker2(&n3);
walker2.walk();
```