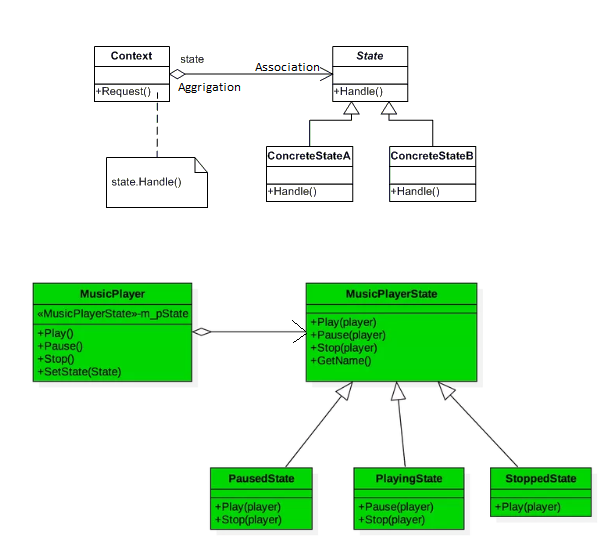
*State Pattern*

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

UML class diagram

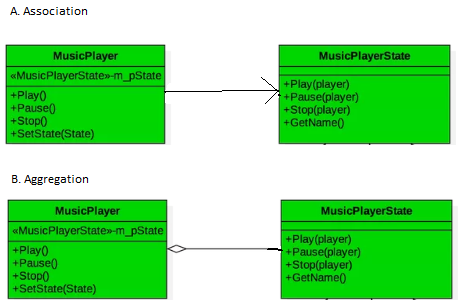


Participants

The classes and objects participating in this pattern are:

* **Context**  **(Account)**
  + defines the interface of interest to clients
  + Maintains an instance of a Concrete State subclass that defines the current state.
* **State**  **(State)**
  + defines an interface for encapsulating the behavior associated with a particular state of the Context.
* **Concrete State**  **(RedState, SilverState, GoldState)**
  + each subclass implements a behavior associated with a state of Context

Discussion about relationships: MusicPlayer & MusicPlayerState



We often confuse with both relationship, which one is correct, let’s see one by one. Find the below code.

class MusicPlayerState;

class MusicPlayer {

public:

MusicPlayer();

virtual ~MusicPlayer();

void Play();

void Pause();

void Stop();

void SetState(MusicPlayerState \*state);

private:

MusicPlayerState \* m\_pState;

};

class MusicPlayerState {

public:

MusicPlayerState(std::string name);

virtual ~MusicPlayerState();

virtual void Play(MusicPlayer \* player)=0;

virtual void Pause(MusicPlayer \* player)=0;

virtual void Stop(MusicPlayer \* player)=0;

virtual std::string GetName()=0;

Protected:

std::string m\_name;

};

Association:

* MusicPlayer has MusicPlayerState, which is not created inside MusicPlayer class,

So both are independent.

* Recall this statement “*The lifetime of the instances of the two classes is independent of each other and there is no ownership between two classes called* ***Association***”.

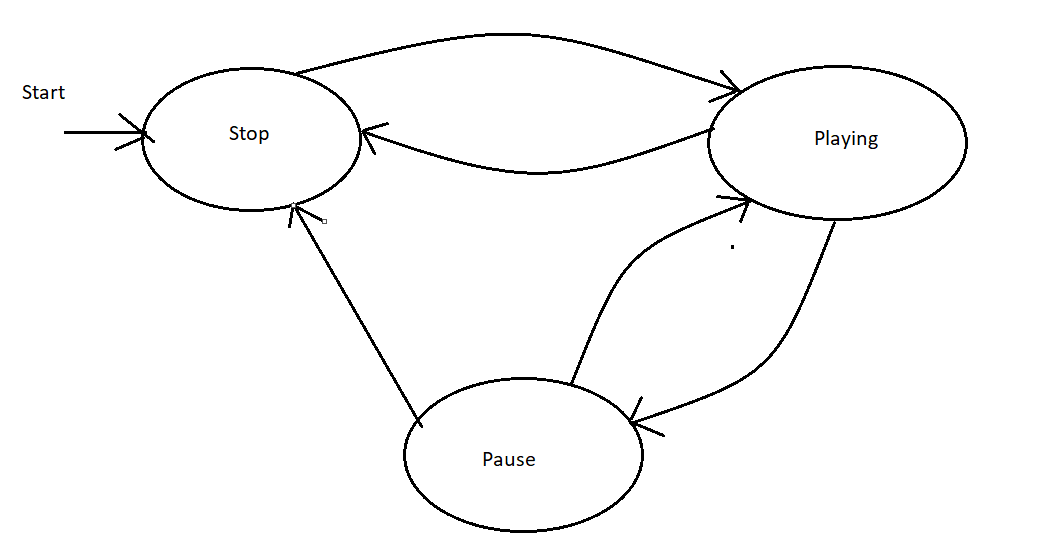
Aggregation:

* Aggregation is a variant of the ***has A*** association relationship or we can say it is more specific then association. And it has logical ownership associated between two classes.

***Purely based on my observation:***

Now it seems like both relationship are correct. Here aggregation is more precise relation then association, because Music player has logical ownership on MusicPlayerState (not a hard ownership otherwise relationship becomes composite). But still more abstract form association is also correct.

State Diagram of above program:



Codes:

class StoppedState : public MusicPlayerState {

public:

StoppedState(std::string nm);

virtual ~StoppedState();

virtual void Play(MusicPlayer \* player);

virtual void Stop(MusicPlayer \* player);

virtual void Pause(MusicPlayer \* player);

virtual std::string GetName();

};

StoppedState::StoppedState(std::string nm) {

name = nm;

}

StoppedState::~StoppedState() {}

void StoppedState::Play(MusicPlayer \* player) {

player->SetState(new PlayingState("Playing"));

}

void StoppedState::Stop(MusicPlayer \* player) {

std::cout << "Illegal state transition from " << GetName().c\_str() << " to stoped\n";

}

void StoppedState::Pause(MusicPlayer \* player) {

std::cout << "Illegal state transition from " << GetName().c\_str() << " to Poused\n";

}

std::string StoppedState::GetName() {

return name;

}

Similarly code for other state like PlayingState and PausedState.

***MusicPlayer.cpp client code.***

#include "MusicPlayer.h"

#include "StoppedState.h"

#include "PlayingState.h"

#include "PausedState.h"

MusicPlayer::MusicPlayer() //Initial State.

: m\_pState(new StoppedState("Stoped")) {

}

MusicPlayer::~MusicPlayer() {

delete m\_pState;

}

void MusicPlayer::Play() {

m\_pState->Play(this);

}

void MusicPlayer::Pause() {

m\_pState->Pause(this);

}

void MusicPlayer::Stop() {

m\_pState->Stop(this);

}

void MusicPlayer::SetState(MusicPlayerState \*state)

{

std::cout << "changing from " << m\_pState->GetName().c\_str() << " to ";

delete m\_pState;

m\_pState = state;

std::cout << m\_pState->GetName().c\_str() << " state\n";

}

//Driver Code.

int main()

{

MusicPlayer player;

player.Play();

player.Play(); //Illigal transition

player.Pause();

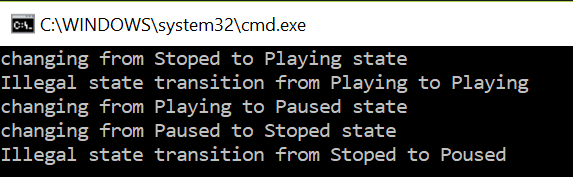
player.Stop();

player.Pause(); //Illigal transition

getchar();

return 0;

}

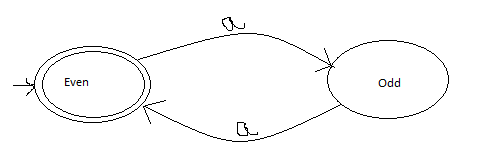


*Finite state Machine Example:*

Construct a finite machine which accepts the even number of a string.

{^,aa,aaaa,aaaaaa…..}

State Diagram.



FiniteMachine Client code.

class FiniteMachine

{

public:

FiniteMachine(char \*);

~FiniteMachine();

void setState(FiniteState\*);

bool getAccepted();

void startProcessing();

private:

FiniteState \*state;

char \*str;

short index;

};

class FiniteState

{

public:

virtual void even() = 0;

virtual void odd() = 0;

virtual std::string getName() = 0;

protected:

std::string name;

};

Relationship: FiniteMachine has FiniteState.

But here story is little bit different. Let’s see the FiniteMachine.cpp code

FiniteMachine::FiniteMachine( char \*tmp) {

str = tmp;

index = 0;

state = nullptr;

}

FiniteMachine::~FiniteMachine() {

delete state;

}

void FiniteMachine::setState(FiniteState \*stat) {

if (state == nullptr) {

std::cout << "Moving state : ^ to " << stat->getName() << "\n";

}

else std::cout << "Moving state :" << state->getName() << " to " << stat->getName() << "\n";

delete state;

state = stat;

}

bool FiniteMachine::getAccepted() {

std::string s = "Even";

if (s == state->getName()) {

return true;

}

else return false;

}

void FiniteMachine::startProcessing() {

FiniteState \*fs;

short len = strlen(str);

while (len > index) {

if (index % 2 == 0) {

fs = new OddState("Odd");

index++;

fs->odd();

}

else {

fs = new EvenState("Even");

index++;

fs->even();

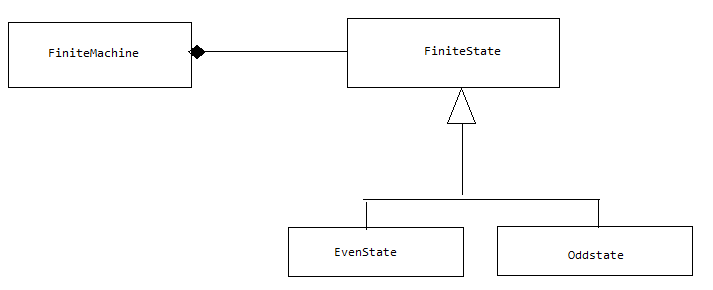
}

setState(fs);

}

}

Here hard dependency (Complete owner ship) between FiniteMachine & FiniteState. FiniteMachine only creates the FiniteState inside the startProcessing() method, so if FiniteMachine has destroyed then its corresponding states also destroy. This kind of relationship called Composition.



class OddState :public FiniteState

{

public:

OddState(std::string stateVal);

void even();

void odd();

std::string getName();

private:

};

class EvenState:public FiniteState

{

public:

EvenState(std::string stateVal);

void even();

void odd();

std::string getName();

private:

};

***EvenState.cpp***

EvenState::EvenState(std::string stateVal) {

name = stateVal;

}

void EvenState::even() {

std::cout << "Machine is in even state, ";

}

void EvenState::odd() {

std::cout << "Invalid state transition\n";

}

std::string EvenState::getName() {

return name;

}

***OddState.cpp***

OddState::OddState(std::string stateVal) {

name = stateVal;

}

void OddState::even() {

std::cout << "Invalid state transition\n";

}

void OddState::odd() {

std::cout << "Machine is in odd state, ";

}

std::string OddState::getName() {

return name;

}

Main Driver

void main() {

FiniteMachine machine("aaaaaaaa");

machine.startProcessing();

if (machine.getAccepted()) {

std::cout << "String is even length\n";

}

getchar();

}

