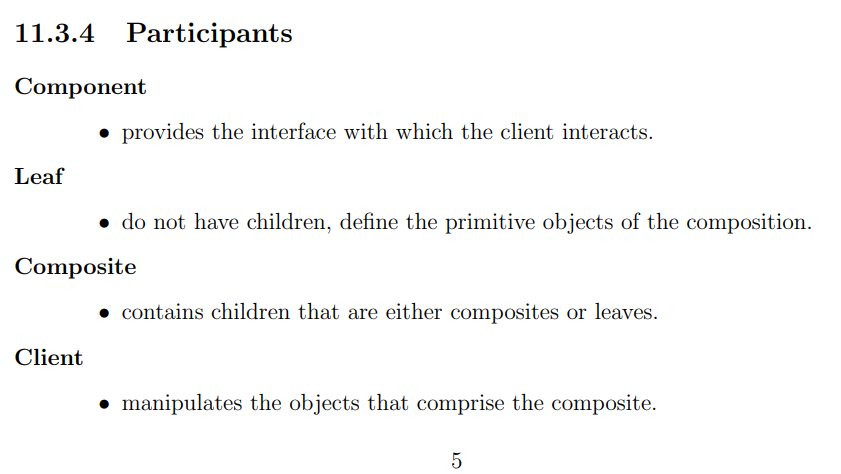
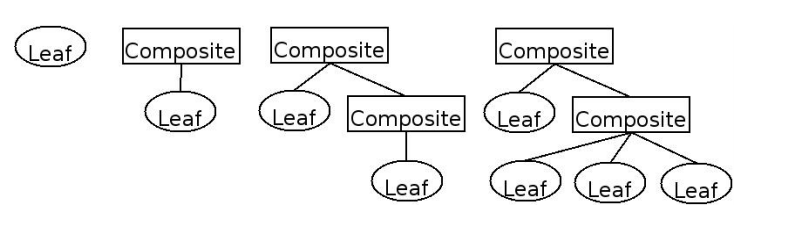
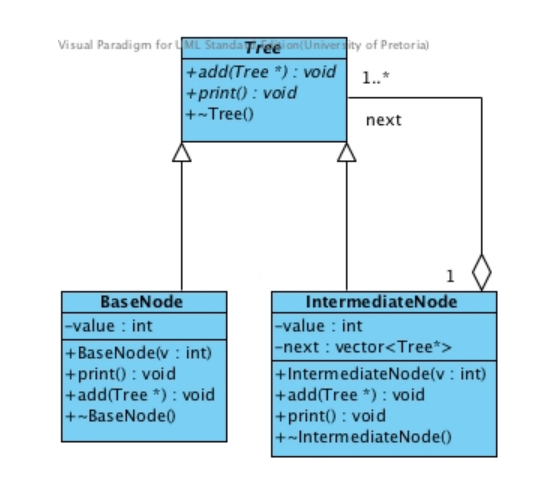
**Composite**







Code:

#include <iostream>

#include <vector>

using namespace std;

class Tree { //component

public:

    virtual void add(Tree\*) = 0;

    virtual void print() = 0;

    virtual ~Tree() {}; // Added

};

class BaseNode : public Tree { //leaf

public:

    BaseNode(int v) : value(v) {};

    virtual void print() {

        cout << " " << value << " ";

    };

    virtual void add(Tree\*) {};

    virtual ~BaseNode() {}; // Added

private:

    int value;

};

class IntermediateNode : public Tree { //composite

public:

    IntermediateNode(int v) : value(v) {};

    virtual void add(Tree\*);

    virtual void print();

    virtual ~IntermediateNode(); // Added

private:

    int value;

    vector<Tree\*> next; //holds leaves and/or other composites

};

void IntermediateNode::add(Tree\* t){

    next.push\_back(t);

}

void IntermediateNode::print(){

    cout << "-" << value << "[";

    vector<Tree\*>:: iterator it;

    for (it = next.begin(); it != next.end(); ++it)//calling print on each of

        (\*it)->print(); the elements in the vector

    cout << "]";

}

IntermediateNode::~IntermediateNode(){

  vector<Tree\*>:: iterator it;

  for (it = next.begin(); it != next.end(); ++it)//self destructs all its

    delete \*it; children so it doesn’t get done in

} main

int main(){

    Tree\* t = new IntermediateNode(10);

    Tree\* b = new BaseNode(5);

    t->add(new BaseNode(5)); // anonymous allocation

    Tree\* l1 = new IntermediateNode(20);

    l1->add(new BaseNode(67)); // anonymous allocation

    l1->add(new BaseNode(20)); // anonymous allocation

    t->add(l1);

    t->print();

    cout<<endl;

    // deallcoate memory in reverse order of allocation

//   delete l1;  // Linked into the tree.

   delete b; // Not linked into Tree t and therefore needs to be deleted separately

   delete t;

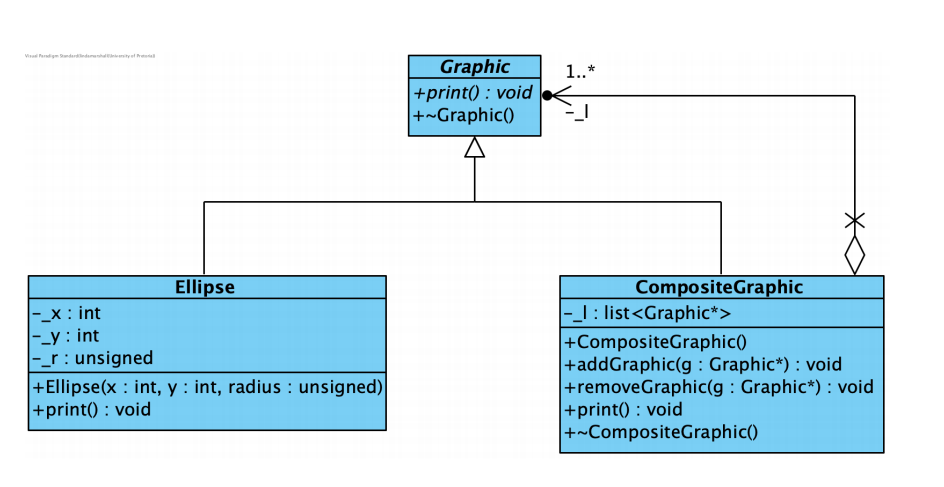
   // This does not delete the anonymous allocations -> Deletion needs to be done by composite

   // Implication is that the composite must implement destructors and that the base class destructor

   //    MUST be virtual

    return 0;

}



#include <iostream>

#include <list>

class Graphic { //component

public:

    /// Print out the Graphic

    virtual void print() = 0;

    // Will ensure well-behaved deletion if no anonymous objects.

    virtual ~Graphic(){

        std::cout << "Deleting" << std::endl;

    };

};

class CompositeGraphic: public Graphic { //composite

public:

    CompositeGraphic() : Graphic(), \_l() {}

    /// Add a child

    void addGraphic(Graphic\* g) {

        \_l.push\_back(g);

    }

    /// Remove a child

    void removeGraphic(Graphic\* g);

    void print(){

        // for each child ...

        for (std::list<Graphic\*>::iterator it =

                 \_l.begin();

                 it != \_l.end();

                 ++it)

            (\*it)->print(); // ... print it

    }

    // Added so that delete is called for all children. This will delete anonymous objects as well.

    ~CompositeGraphic() {

        for (std::list<Graphic\*>::iterator it =

             \_l.begin();

             it != \_l.end();

             ++it)

            delete \*it;

    }

private:

    /// children

    std::list<Graphic\*> \_l;

};

class Ellipse: public Graphic {//leaf

public:

    /// Build an ellipse with the specified

    // coordinates and radius

    Ellipse(int x, int y, unsigned radius)

    : Graphic(), \_x(x), \_y(y), \_r(radius) {}

    virtual void print() {

        std::cout << "Ellipse("

        << \_x << ", "

        << \_y << ", "

        << \_r << ")"

        << std::endl;

    }

private:

    int \_x;

    int \_y;

    unsigned \_r;

};

int main(){

  Ellipse\* e1 = new Ellipse(42, 51, 69);

  Ellipse\* e2 = new Ellipse(16, 64, 86);

  Ellipse\* e3 = new Ellipse(1, 33, 7);

  CompositeGraphic\* g1 = new CompositeGraphic();

  CompositeGraphic g2;

  g1->addGraphic(e1);

  g1->addGraphic(e2);

  g2.addGraphic(e3);

  g2.addGraphic(g1);

  std::cout<<"g1 = "<<std::endl;

  g1->print();

  std::cout<<"g2 = "<<std::endl;

  g2.print();

  /\*

    g2 is on the stack and therefore not necessary to explicitely call delete for the structure.

   \*/

  return 0;

}

/\*

 g2

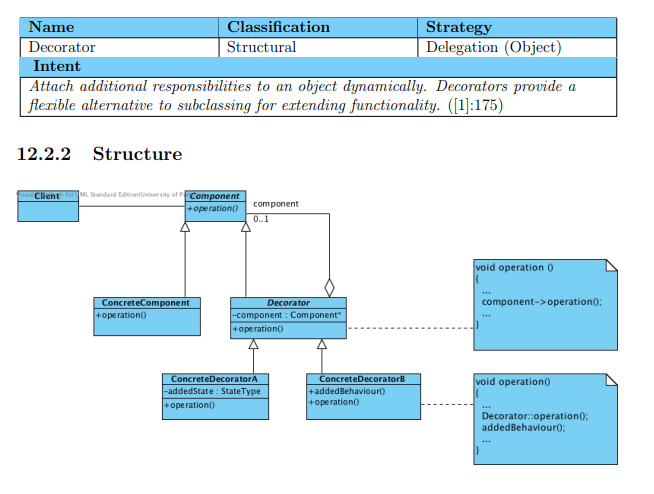
 |-1-> e3

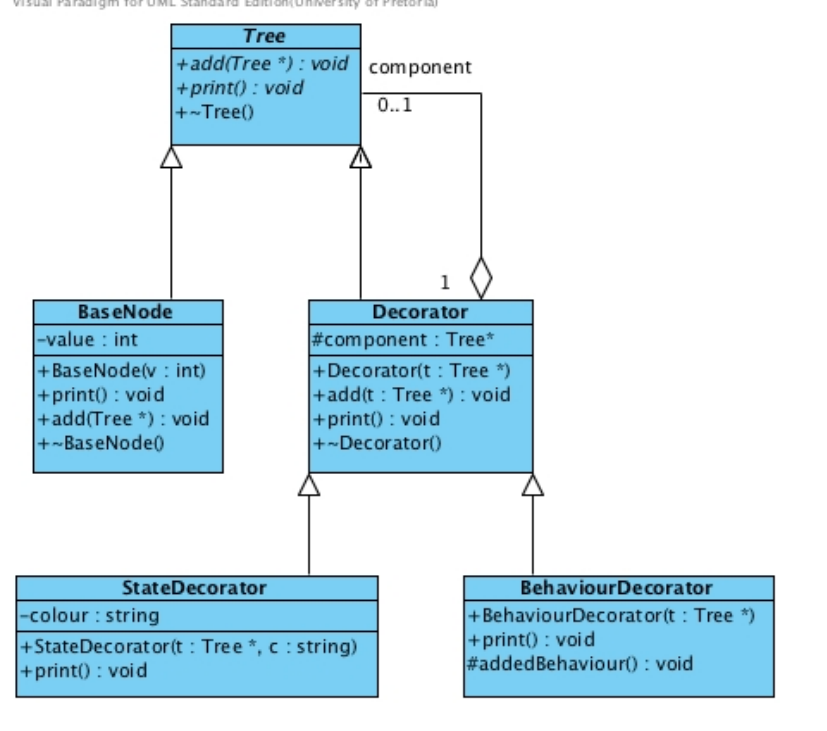
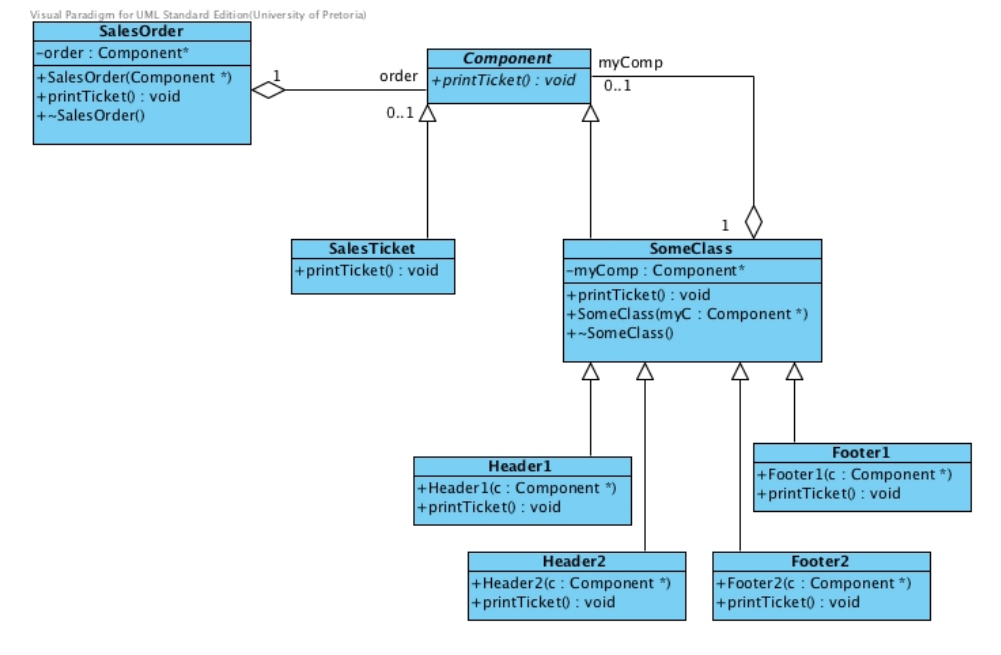
 |-2-> g1

       |-2.1-> e1

       |-2.2-> e2

 \*/

Decorator



#include <iostream>

using namespace std;

class Component {

  public:

    virtual void printTicket() = 0;

  virtual ~Component() {};

};

class SalesTicket: public Component {//concrete component

  public:

    void printTicket();

  virtual ~SalesTicket() {};

};

void SalesTicket::printTicket() {

  cout<<"List of items purchased"<<endl;

}

class SomeClass: public Component {//decorator

  public:

    virtual void printTicket();

    SomeClass(Component\* myC);

    // Need to add a destructor - MUST be virtual

    virtual ~SomeClass();

  private:

    Component \*myComp;

};

SomeClass::SomeClass(Component\* myC) {

  myComp = myC;

}

void SomeClass::printTicket() {

  if (myComp)

    myComp->printTicket();

}

SomeClass::~SomeClass(){

  delete myComp;

}

class Header1: public SomeClass {//concrete decorator

  public:

    Header1(Component\* c);

    void printTicket();

};

Header1::Header1(Component\* c) : SomeClass(c) { }

void Header1::printTicket() {

  cout<<"Welcome to the Crazy Zone"<<endl;

  SomeClass::printTicket();

}

class Header2: public SomeClass {//concrete decorator

  public:

    Header2(Component\* c);

    void printTicket();

};

Header2::Header2(Component\* c) : SomeClass(c) { }

void Header2::printTicket()

{

  cout<<"Shopping at the Crazy Zone"<<endl;

  SomeClass::printTicket();

}

class Footer1: public SomeClass {//concrete decorator

  public:

    Footer1(Component\* c);

    void printTicket();

};

Footer1::Footer1(Component\* c) : SomeClass(c) { }

void Footer1::printTicket() {

  SomeClass::printTicket();

  cout << "It was a pleasure doing" <<

          " business with you"<<endl;

}

class Footer2: public SomeClass {//concrete decorator

  public:

    Footer2(Component\* c);

    void printTicket();

};

Footer2::Footer2(Component\* c) : SomeClass(c) { }

void Footer2::printTicket() {

  SomeClass::printTicket();

  cout << "Enjoy your day"<<endl;

}

class SalesOrder {

  public:

    SalesOrder(Component\*);

    void printTicket();

    ~SalesOrder();

  private:

    Component\* order;

};

SalesOrder::SalesOrder(Component\* c) : order(c) {}

void SalesOrder::printTicket() {

  order->printTicket();

}

SalesOrder::~SalesOrder(){

    delete order;

}

int main() {

    SalesOrder\* s = new SalesOrder(new Footer1(

                           new Header1(

                           new SalesTicket)));

      // Note: SalesTicket is being decorated.

    s->printTicket();

    // Destruct all relevant objects

    delete s;

  return 0;

}

/\*

 Example of output:

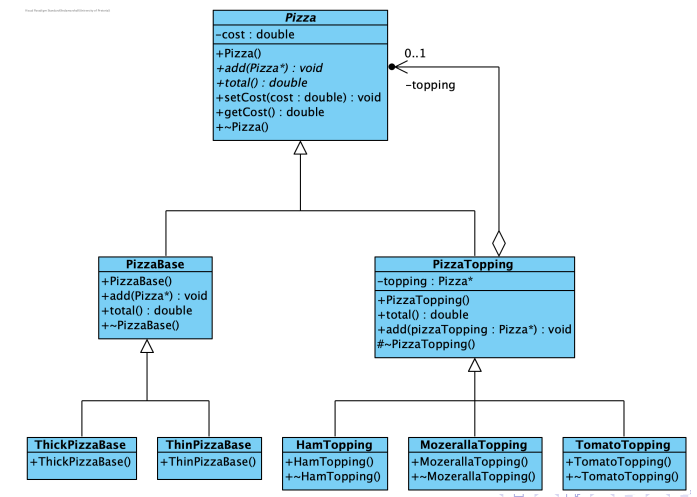
 Welcome to the Crazy Zone

 List of items purchased

 It was a pleasure doing business with you

 \*/

//



//  Pizza.cpp

//

//

//  Created by Linda Marshall on 2017/08/20.

//

//

#include <iostream>

using namespace std;

class Pizza { //component

public:

    Pizza() {

        cost = 0.0;

    };

    virtual void add(Pizza\*) = 0;

    virtual double total() = 0;

    void setCost(double cost) {

        this->cost = cost;

    };

    double getCost() {

        return cost;

    };

    virtual ~Pizza() {} ;

private:

    double cost;

};

class PizzaBase : public Pizza {//concrete component

public:

    PizzaBase() {};

    virtual void add(Pizza\*) {};

    virtual double total() {

        return getCost();

    };

    ~PizzaBase() {};

};

class ThinPizzaBase : public PizzaBase {//concrete component

public:

    ThinPizzaBase() {

        cout << "Creating a thin base" << endl;

        setCost(15.00);

    };

};

class ThickPizzaBase : public PizzaBase {//concrete component

public:

    ThickPizzaBase() {

        cout << "Creating a thick base" << endl;

        setCost(25.00);

    };

};

class PizzaTopping : public Pizza {//decorator

public:

    PizzaTopping() {

        topping = 0;

    };

    virtual double total() {

        // cout << "In PizzaTopping total" << endl;

        if (topping == 0) {

            return getCost();

        } else {

          return getCost() + topping->total();

        }

    };

    virtual void add(Pizza\* pizzaTopping) {

        cout << "In PizzaTopping add" << endl;

        if (topping == 0){

            topping = pizzaTopping;

        } else {

            topping->add(pizzaTopping);

        }

    };

protected:

    ~PizzaTopping() {delete topping; };

private:

    Pizza\* topping;

};

class TomatoTopping : public PizzaTopping {

public:

    TomatoTopping() : PizzaTopping() {

        cout << "Creating tomato topping" << endl;

        setCost(5.00);

    };

    ~TomatoTopping() {};

};

class MozerallaTopping : public PizzaTopping {

public:

    MozerallaTopping() : PizzaTopping() {

        cout << "Creating Mozeralla topping" << endl;

        setCost(10.00);

    };

    ~MozerallaTopping() {};

};

class HamTopping : public PizzaTopping {

public:

    HamTopping() : PizzaTopping() {

        cout << "Creating ham topping" << endl;

        setCost(15.00);

    };

    ~HamTopping() {};

};

int main() {

    Pizza\* myPizza;

    myPizza = new TomatoTopping();

    myPizza->add(new MozerallaTopping());

    myPizza->add(new HamTopping());

    myPizza->add(new HamTopping());

    myPizza->add(new ThickPizzaBase());

    cout << "Cost = " << myPizza->total() << endl;

    delete myPizza;

    return 0;

}

/\* Program output:

 Creating tomato topping

 Creating Mozeralla topping

 In PizzaTopping add

 Creating ham topping

 In PizzaTopping add

 In PizzaTopping add

 Creating ham topping

 In PizzaTopping add

 In PizzaTopping add

 In PizzaTopping add

 Creating a thick base

 In PizzaTopping add

 In PizzaTopping add

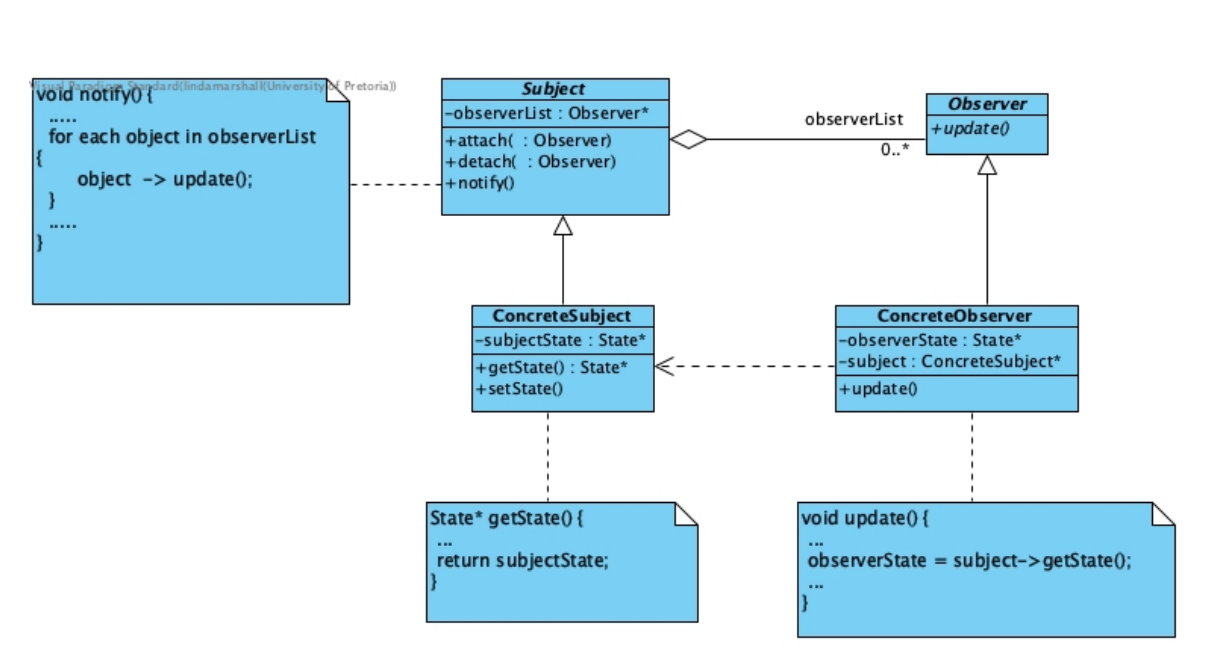
 In PizzaTopping add

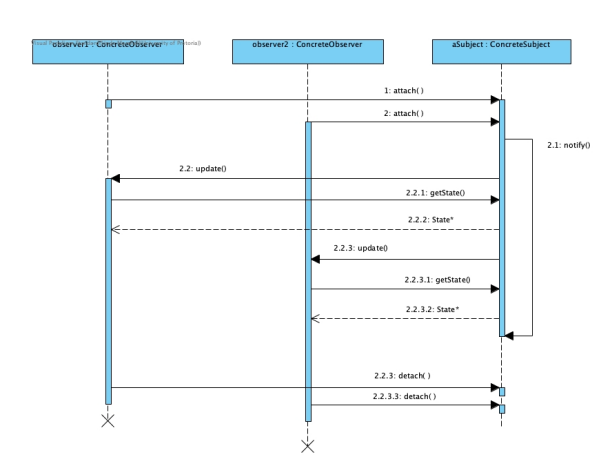
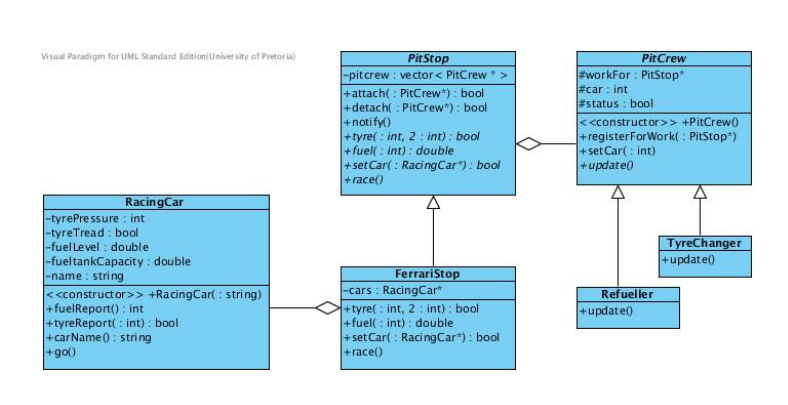
 In PizzaTopping add

 Cost = 70

 \*/

Observer





Examples:

Examples:

#include <iostream>

#include <string>

#include <vector>

using namespace std;

class RacingCar {

  public:

    RacingCar(string);

    int fuelReport();

    bool tyreReport(int);

    string carName();

    void go();

  private:

    int   tyrePressure[4];

    bool   tyreTread[4];

    double fuelLevel;

    double fueltankCapacity;

    string name;

};

class PitCrew;

class PitStop { // Subject

  public:

    bool attach(PitCrew\*);   // register

    bool detach(PitCrew\*);   // deregister

    void notify();

    virtual bool tyre(int,int) = 0;

    virtual double fuel(int) = 0;

    virtual bool setCar(RacingCar\*) = 0;

    virtual void race() = 0;

  private:

    vector<PitCrew\*> pitcrew;

};

// An observer

class PitCrew {

  public:

    PitCrew() : car(1), workFor(0), status(0) {};

    void registerForWork(PitStop\*);

    void setCar(int);

    virtual void update() = 0;

    // need to refuel and change tyres

  protected:

    PitStop\* workFor;

    int car;

    bool status;

};

bool PitStop::attach(PitCrew\* person) {

  pitcrew.push\_back(person);

  person->registerForWork(this);

  return true;

};

bool PitStop::detach(PitCrew\* person) {

  bool found = false;

  vector<PitCrew\*>::iterator it = pitcrew.begin();

  while ((it != pitcrew.end()) && (! found)) {

    if (\*it == person) {

      found = true;

      pitcrew.erase(it);

    }

    ++it;

  }

  return found;

}

void PitStop::notify(){

  vector<PitCrew\*>::iterator it = pitcrew.begin();

  for (it = pitcrew.begin(); it != pitcrew.end(); ++it){

    (\*it)->update();

  }

}

// Helper function

void printWorkshopStatus(PitStop\* p) {

  cout << "Fuel for car 1 = " << p->fuel(1) << endl;

  for (int i = 1; i <= 4; i++) {

    cout << "Tyre for car 1, tyre " << i << " = " << p->tyre(1,i) << endl;

  }

  cout << "Fuel for car 2 = " << p->fuel(2) << endl;

  for (int i = 1; i <= 4; i++) {

    cout << "Tyre for car 2, tyre " << i << " = " << p->tyre(2,i) << endl;

  }

}

class FerrariStop : public PitStop { // concrete subject

  public:

    bool tyre(int,int);

    double fuel(int);

    bool setCar(RacingCar\*);

    void race();

  private:

    RacingCar\* cars[2];

};

bool FerrariStop::tyre(int car, int tyre) { // Need some error checking here

  return cars[car-1]->tyreReport(tyre-1);

}

double FerrariStop::fuel(int car) { // Need some error checking here

  return cars[car-1]->fuelReport();

}

bool FerrariStop::setCar(RacingCar\* car) {

  static int carId = 0;

  if (carId < 2) {

    cars[carId] = car;

    carId++;

    return true;

  }

  return false;

}

void FerrariStop::race() {

  int input;

  cout << "Type in a number [0 stops]:";

  cin >> input;

  while (input != 0) {

    if ((input % 2) == 0) {

      cars[0]->go();

    } else {

      cars[1]->go();

    }

    printWorkshopStatus(this);

    notify();

    cout << "Type in a number [0 stops]";

    cin >> input;

  }

}

void PitCrew::registerForWork(PitStop\* employer) {

  workFor = employer;

}

void PitCrew::setCar(int c) {

  car = c;

}

// Concrete observer1

class TyreChanger : public PitCrew {

  public:

    virtual void update();

};

void TyreChanger::update() {

  cout << "Tyre changer for car " << car << " status is " << status << endl;

  if (status == 0) {

    cout << "Check tyre status" << endl;

    bool tyreStatus = false;

    for (int i = 1; i <= 4; i++)

      tyreStatus = tyreStatus && workFor->tyre(car,i);

    if (tyreStatus) {

      status = 1;

      cout << "Need to change all tyres" << endl;

    }

  } else

    status = 0;

}

// Concrete observer2

class Refueller : public PitCrew {

  public:

    virtual void update();

};

void Refueller::update() {

  cout << "Refeuller for car " << car << " status is " << status << endl;

  if (status == 0) {

    cout << "Check fuel status" << endl;

    double fuelStatus = workFor->fuel(car);

    cout << "  fuel status is: " << fuelStatus << endl;

    if (fuelStatus < 20) {

      status = 1;

      cout << "Need to add fuel" << endl;

    }

  } else

    status = 0;

}

RacingCar::RacingCar(string n) : name(n) {

  for (int i = 0; i < 4; i++) {

    tyrePressure[i] = 4;

    tyreTread[i] = true;

  }

  fueltankCapacity = 100;

  fuelLevel = 100;

}

int RacingCar::fuelReport() {

  return fuelLevel / fueltankCapacity \* 100;

}

bool RacingCar::tyreReport(int tyre) {

  return tyrePressure[tyre] && tyreTread[tyre];

}

string RacingCar::carName() {

  return name;

}

void RacingCar::go() {

  int input;

  cout << "Type in any value: " << endl;

  cin >> input;

  if ((input % 2) == 0) {

    // Do the tyres

    if ((input % 3) == 0) {

      tyreTread[input%4] = false;

    } else {

      tyrePressure[input%4] = false;

    }

  } else {

    // Do the fuel

    fuelLevel -= 5;

  }

}

int main() {

  RacingCar\* car[2];

  car[0] = new RacingCar("Ferrari One");

  car[1] = new RacingCar("Ferrari Two");

  PitStop\* ferrariWorkshop = new FerrariStop();

//  FerrariStop\* ferrariWorkshop = new FerrariStop();

  ferrariWorkshop->setCar(car[0]);

  ferrariWorkshop->setCar(car[1]);

  printWorkshopStatus(ferrariWorkshop);

/\*

  for (int i = 0; i < 10; i++) {

    car[0]->go();

    printWorkshopStatus(ferrariWorkshop);

  }

\*/

  PitCrew\* refueller = new Refueller();

  refueller->setCar(2);

  ferrariWorkshop->attach(refueller);

  PitCrew\* tyreMech = new TyreChanger();

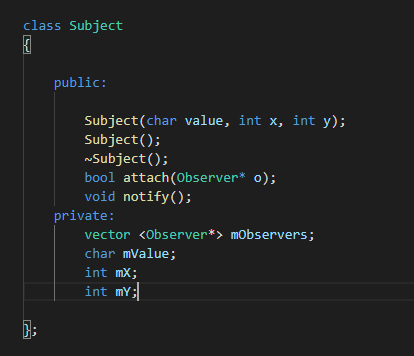
  ferrariWorkshop->attach(tyreMech);

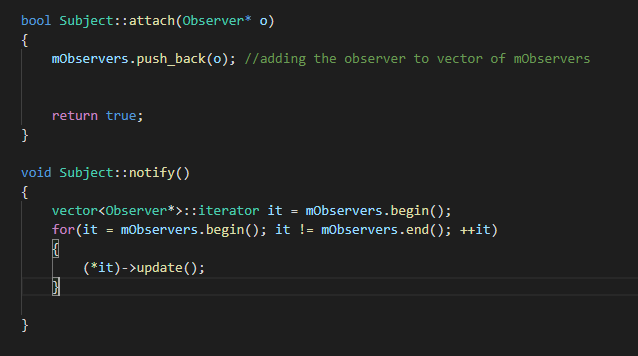
  ferrariWorkshop->race();

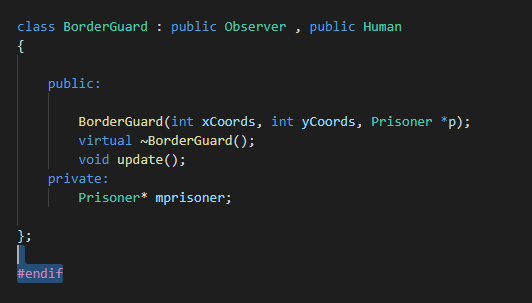
  return 0;

}

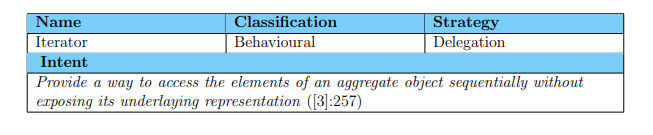
Prison example:

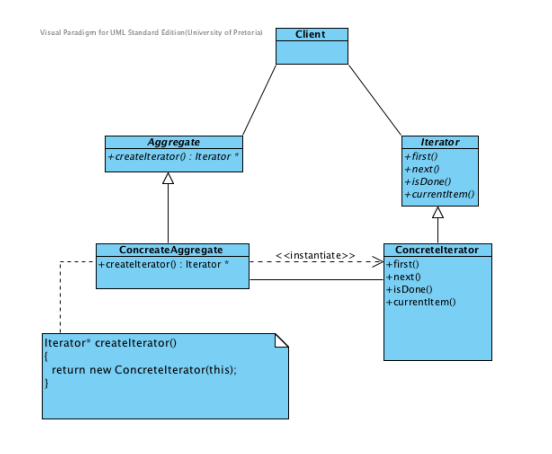


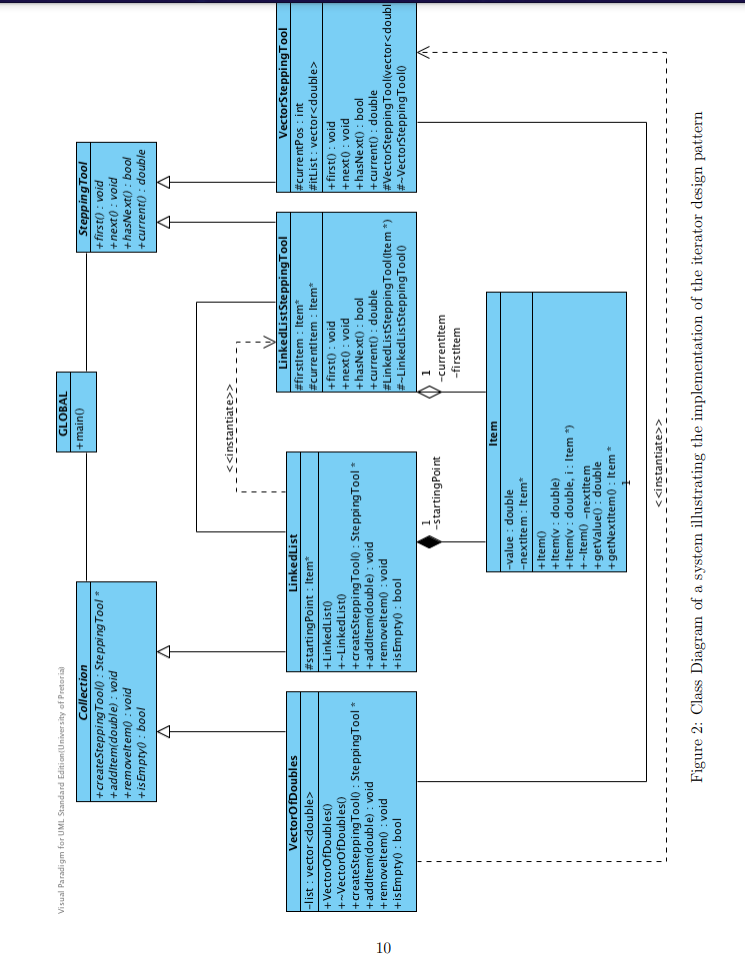


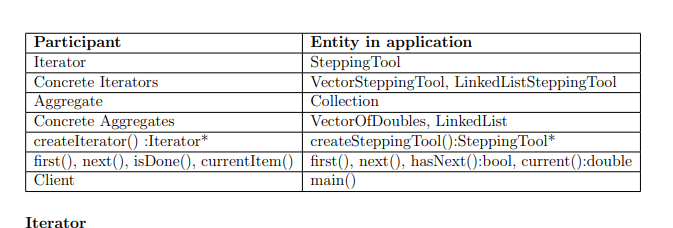


Iterator:

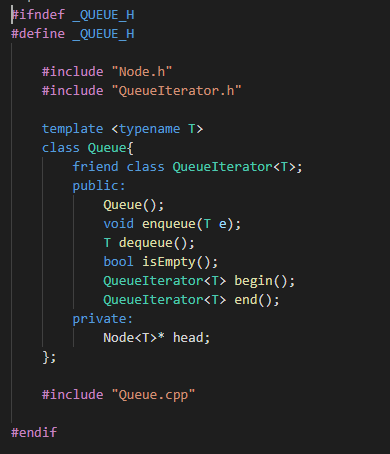


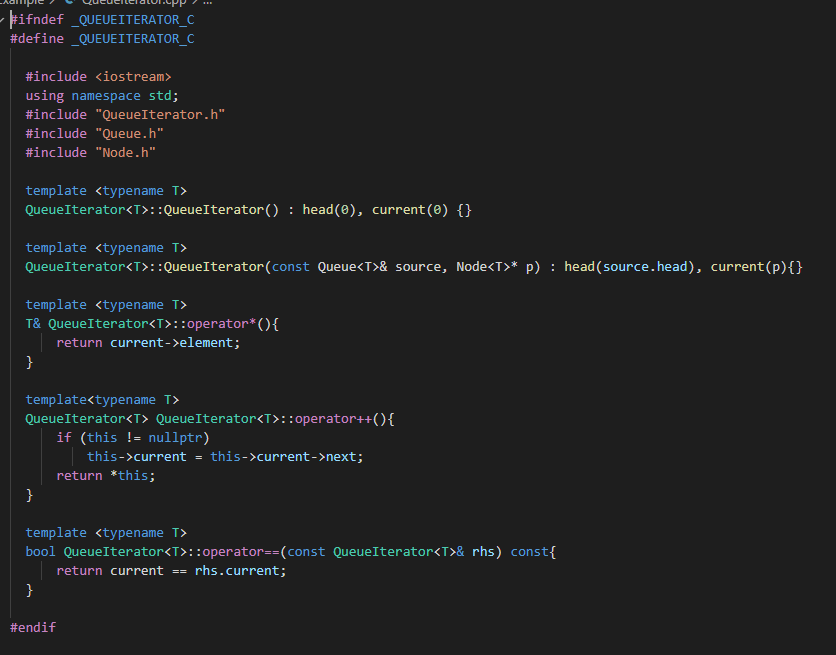




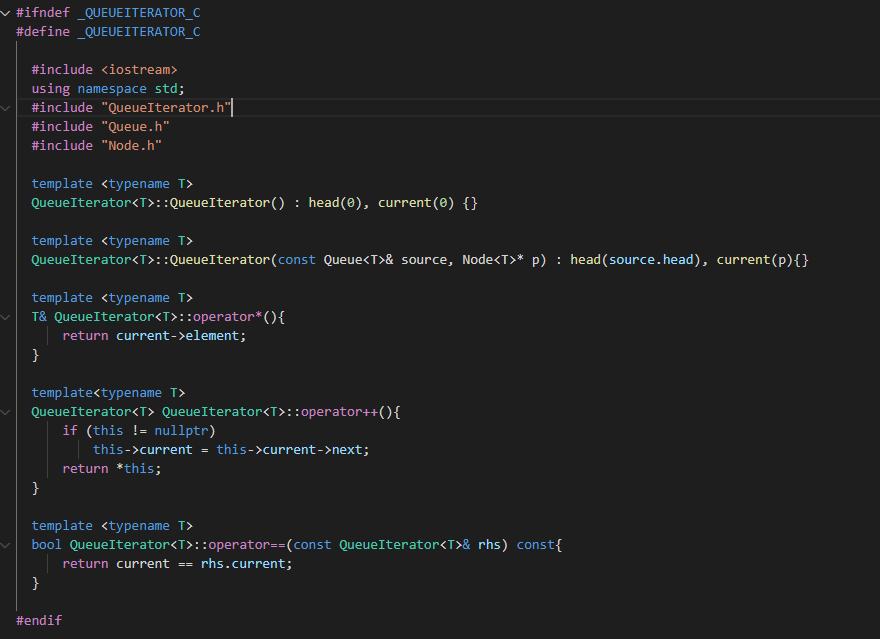


Example:

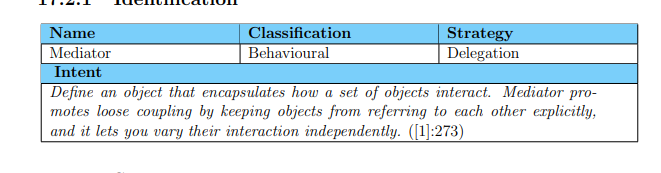


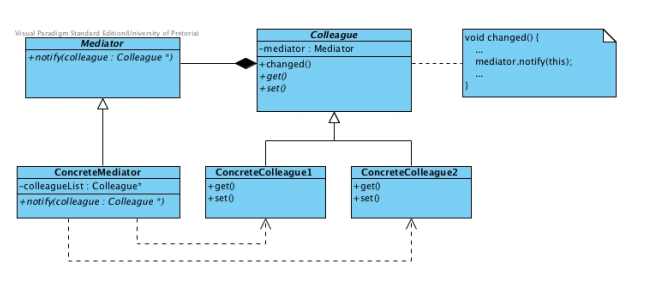


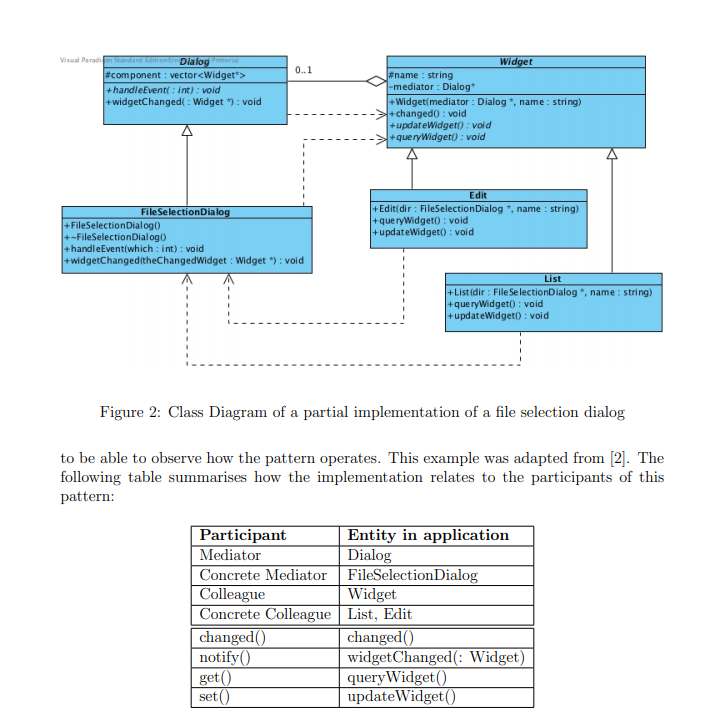




Mediator:







Examples

