*Iterator Pattern*

**Definition:** The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The IP allows traversal of the elements of an aggregates without exposing the underlying implementation.

To understand the IP, let’s discuss first, why we need it?

Let’s suppose there are two restaurants in which, one servers only breakfast items and other serves Dinner & Launch items. So both have their corresponding menu, but menu implements are different in both of restaurant.

class MenuItem

{

public:

MenuItem() {}

MenuItem(string nm,string disc,float prc,bool var):

name(nm),discription(disc),price(prc),isVeg(var){}

~ MenuItem(){}

string getName() { return name; }

string getDiscription() { return discription; }

float getPrice() { return price; }

bool isVegeterian() { return isVeg; }

private:

string name;

string discription;

float price;

bool isVeg;

};

//Having Breakfast Item, menu is a list type.

class RestaurentAMinu

{

public:

class RestaurentAMinu() {

addItem("Idli","Soft Rice Idli with sambhar",20.50,true);

addItem("Puri", "Made with aata with sabji", 25.26, true);

addItem("BreadAmlet", "Bread with two eggs", 10.23, false);

addItem("Paratha", "Aalo paratha with chatni", 12.23, true);

addItem("Chicken Paratha", "Chicken paratha with curd", 35.23, false);

}

list <MenuItem\*> getMenuItems() {

return menuList;

}

private:

list <MenuItem\*> menuList;

void addItem(string nm, string disc, float prc, bool var) {

MenuItem \*mItem=new MenuItem(nm, disc, prc, var);

menuList.push\_back(mItem);

}

};

//Having Launch & dinner Item, menu is array type

class RestaurentBMinu

{

public:

int nbrOfItems;

RestaurentBMinu() {

addItem("Roti", "Simple Roti", 3.00, true);

addItem("Rice", "Simple Rice", 10.00, true);

addItem("Daal", "Daal Tadka", 23.20, true);

addItem("Chicken kadi", "With Thick gravi", 50.50, false);

}

MenuItem\* getMenuItems() {

return menuArr;

}

private:

MenuItem menuArr[MAX\_ITEM];

void addItem(string nm, string disc, float prc, bool var) {

MenuItem mItem(nm, disc, prc, var);

if (nbrOfItems < MAX\_ITEM) {

menuArr[nbrOfItems++] = mItem;

}

else {

cout << "Insufficinet Space\n\n\n";

}

}

};

Waitress will provide below services.

She will take order for both of the restaurant and provide the services to customer.

class Waitress {

RestaurentAMinu r1;

RestaurentBMinu r2;

public:

void printMenu();

void printBreakfastMenu();

void printLaunchDinnerMenu();

void printVegMenu();

void printNonVegMenu();

};

void Waitress::printMenu() {

In order to print menu, need to fetch the Items form both of the restaurant.

And need to iterate the below list1 & list2. It required multiple loops.

Also need to know how they are implemented (internal details like list, array

etc implementation of menu Items) which is not good.

list<MenuItem\*> list1 = r1.getMenuItems(); //List Implemention.

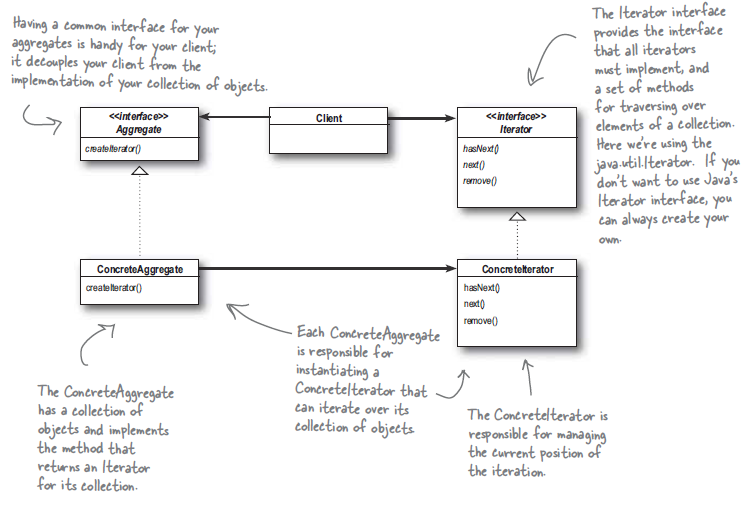
MenuItem \*list2 = r2.getMenuItems(); //Array Implemention.

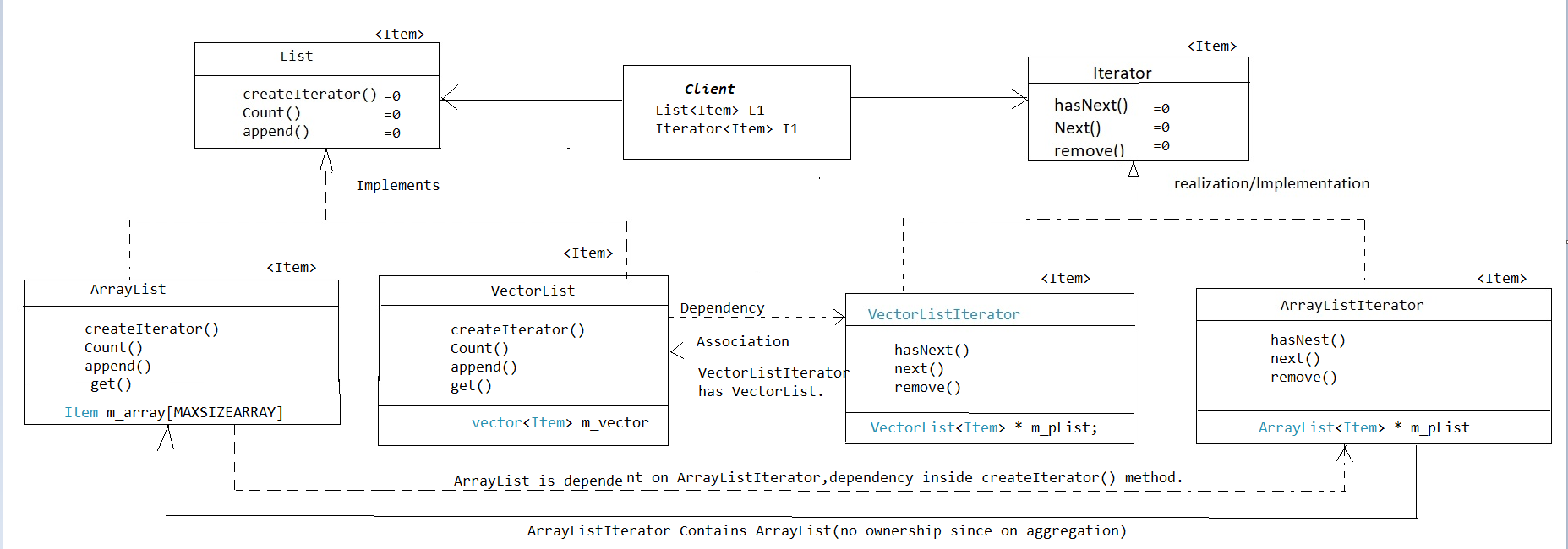
It is always batter to allow implementing some interface thru which

Can minimize the concrete reference in the waitress code; also don’t reveal internal implementation details.

}

Solution is **Iterator Pattern.**

****



**List.h**

template <class Item>

class List {

public:

virtual Iterator<Item> \* CreateIterator() const = 0;

virtual unsigned int Count() const = 0;

virtual void Append(Item item) = 0;

};

**ArrayList.h**

const unsigned int MAXSIZEARRAY = 200;

template <class Item>

class Iterator;

template <class Item>

class ArrayList : public List<Item> {

public:

ArrayList(): m\_currentSize(0){}

virtual ~ArrayList() {}

virtual Iterator<Item> \* CreateIterator() const {

return new ArrayListIterator<Item>(this);

}

virtual unsigned int Count() const {

return m\_currentSize;

}

virtual void Append(Item item) {

if(m\_currentSize < MAXSIZEARRAY)

{

m\_array[m\_currentSize] = item;

m\_currentSize++;

}

}

virtual const Item& Get(unsigned int index) const

{

return m\_array[index];

}

private:

Item m\_array[MAXSIZEARRAY];

unsigned int m\_currentSize;

};

**StdVectorList.h**

template <class Item>

class Iterator;

template <class Item>

class StdVectorList : public List<Item> {

public:

StdVectorList(){}

virtual ~StdVectorList(){}

virtual Iterator<Item> \* CreateIterator() const {

return new StdVectorListIterator<Item>(this);

}

virtual unsigned int Count() const {

return m\_vector.size();

}

virtual void Append(Item item) {

m\_vector.push\_back(item);

}

virtual const Item& Get(unsigned int index) const

{

return m\_vector.at(index);

}

private:

std::vector<Item> m\_vector;

};

**Iterator.h**

template <class Item>

class Iterator {

public:

virtual void First() = 0;

virtual void Next() = 0;

virtual bool IsDone() const = 0;

virtual Item CurrentItem() const = 0;

};

**ArrayListIterator.h**

template <class Item>

class ArrayListIterator : public Iterator<Item> {

public:

ArrayListIterator(const ArrayList<Item> \* list): m\_pList(list), m\_index(0) { }

virtual ~ArrayListIterator(){}

virtual void First() {

m\_index = 0;

}

virtual void Next() {

m\_index++;

}

virtual bool IsDone() const {

return (m\_pList->Count() <= m\_index);

}

virtual Item CurrentItem() const

{

if(IsDone()){

return Item();

//throw IteratorOutOfBounds;

}

return m\_pList->Get(m\_index);

}

private:

const ArrayList<Item> \* m\_pList;

unsigned int m\_index;

};

**StdVectorListIterator.h**

template <class Item>

class StdVectorListIterator : public Iterator<Item> {

public:

StdVectorListIterator(const StdVectorList<Item> \* list) : m\_pList(list),m\_index(0) {}

virtual ~StdVectorListIterator() { }

virtual void First() {

m\_index = 0;

}

virtual void Next() {

m\_index++;

}

virtual bool IsDone() const {

return (m\_pList->Count() <= m\_index);

}

virtual Item CurrentItem() const {

if(IsDone()){

//Return Item()

//throw IteratorOutOfBounds;

}

return m\_pList->Get(m\_index);

}

private:

const StdVectorList<Item> \* m\_pList;

unsigned int m\_index;

};

**Main. Clint Code , here BaseballPlayer is <Item>**

List<BaseballPlayer> \* MakeNewYorkMetsLineup()

{

List<BaseballPlayer> \* newYorkMetsLineup = new StdVectorList<BaseballPlayer>();

newYorkMetsLineup->Append(BaseballPlayer(std::string("Curtis Granderson"), std::string("CF"), 'L'));

newYorkMetsLineup->Append(BaseballPlayer(std::string("Asdrubal Cabrera"), std::string("SS"), 'S'));

newYorkMetsLineup->Append(BaseballPlayer(std::string("Yoenis Cespedes"), std::string("LF"), 'R'));

newYorkMetsLineup->Append(BaseballPlayer(std::string("Jay Bruce"), std::string("RF"), 'L'));

newYorkMetsLineup->Append(BaseballPlayer(std::string("Neil Walker"), std::string("2B"), 'S'));

newYorkMetsLineup->Append(BaseballPlayer(std::string("Lucas Duda"), std::string("1B"), 'L'));

newYorkMetsLineup->Append(BaseballPlayer(std::string("Travis d'Arnaud"), std::string("C"), 'R'));

newYorkMetsLineup->Append(BaseballPlayer(std::string("Jose Reyes"), std::string("SS"), 'S'));

newYorkMetsLineup->Append(BaseballPlayer(std::string("Noah Syndergaard"), std::string("P"), 'L'));

return newYorkMetsLineup;

}

List<BaseballPlayer> \* MakeWashingtonNationalsLineup()

{

List<BaseballPlayer> \* washingtonNationalsLineup = new ArrayList<BaseballPlayer>();

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Adam Eaton"), std::string("CF"), 'L'));

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Wilmer Difo"), std::string("SS"), 'R'));

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Daniel Murphy"), std::string("2B"), 'L'));

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Bryce Harper"), std::string("RF"), 'L'));

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Ryan Zimmerman"), std::string("1B"), 'L'));

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Anthony Rendon"), std::string("3B"), 'R'));

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Matt Wieters"), std::string("C"), 'R'));

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Michael Taylor"), std::string("CF"), 'R'));

washingtonNationalsLineup->Append(BaseballPlayer(std::string("Stephen Strasburg"), std::string("P"), 'S'));

return washingtonNationalsLineup;

}

void PrintLineup(std::string lineupName,

Iterator<BaseballPlayer> \* iterator)

{

std::cout << lineupName << "\n";

unsigned int lineupPosition = 1;

for(iterator->First(); !iterator->IsDone(); iterator->Next())

{

std::cout << lineupPosition++ << ". " << iterator->CurrentItem() << "\n";

}

std::cout << "\n";

}

int main()

{

List<BaseballPlayer> \* newYorkMetsLineup = MakeNewYorkMetsLineup();

Iterator<BaseballPlayer> \* newYorkMetsLineupIterator =

newYorkMetsLineup->CreateIterator();

PrintLineup(std::string("New York Mets Lineup"), newYorkMetsLineupIterator);

delete newYorkMetsLineupIterator;

delete newYorkMetsLineup;

List<BaseballPlayer> \* washingtonNationalsLineup = MakeWashingtonNationalsLineup();

Iterator<BaseballPlayer> \* washingtonNationalsLineupIterator = washingtonNationalsLineup->CreateIterator();

PrintLineup(std::string("Washington Nationals Lineup"), washingtonNationalsLineupIterator);

delete washingtonNationalsLineupIterator;

delete washingtonNationalsLineup;

return 0;

}