Curs 11

- **Containere cu elemente generice**
- **Șabloane de proiectare**
- **STL – Standard Template Library**
  - **containere**
  - **iterator**

Containere cu elemente generice

Creare de containere (liste, multimi, dictionar,etc) care acceptă orice tip de elemente:

- **void ***

```
typedef void* TElem;

class DynamicArray {
public:
	/**
	 * Add an element to the dynamic array to the end of the array
	 * e - is a generic element
	 */
	void add(TElem e);
	/**
	 * Access the element from a given position
	 * poz - the position (poz>=0;poz<size)
	 */
	TElem get(int poz);
```

- **Șabloane**

```
template<typename Element>
class DynamicArray {
public:
	/**
	 * Add an element to the dynamic array to the end of the array
	 * e - is a generic element
	 */
	void addE(Element r);
	/**
	 * Access the element from a given position
	 * poz - the position (poz>=0;poz<size)
	 */
	Element get(int poz);
```

Containere cu elemente generice

Dacă structura de date are nevoie de anumite funcții (egalitate, hashCode, copiere de valori, etc.):

- **pointer la funcții**

```
typedef elem (*copyPtr)(elem&, elem);
typedef int (*equalsPtr)(elem, elem);
```

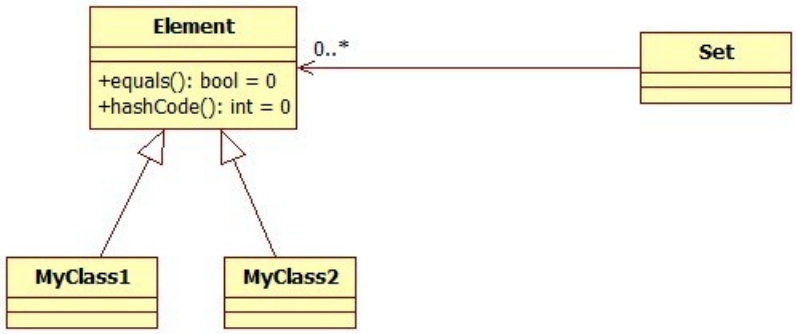
- **supraâncărcare operatori**

```
class Product {
public:
	bool Product::operator==(const Product &other) const {
		return code == other.code;
	}
```

```
template<typename Element>
class Set {
private:
	Element* elems;
	int size;
public:
	Set();
	void add(Element el);
	bool contains(Element el);
	int card() {
		return size;
	}
};
template<typename Element>
Set<Element>::Set() {
	elems = new Element[100];
	size = 0;
}
```

```
template<typename Element>
void Set<Element>::add(Element el) {
	if (contains(el)) {
		return;
	}
	elems[size] = el;
	size++;
}
template<typename Element>
bool Set<Element>::contains(Element el) {
	for (int i = 0; i < size; i++) {
		if (el == elems[i]) {
			return true;
		}
	}
	return false;
}
```

- **creăm o clasă de bază abstractă cu metode virtuale**



Șabloane de proiectare

- Șabloanele de proiectare descriu obiecte, clase si interacțiuni/relații intre ele.Un șablon reprezintă o soluție comună a unei probleme într-un anumit context

- Sunt soluții generale, reutilizabile pentru probleme ce apar frecvent întrun context dat

- Christopher Alexander: "Fiecare şablon descrie o problemă care apare mereu în domeniul nostru de activitate şi indică esenţa soluţiei acelei probleme într-un mod care permite utilizarea soluţiei de nenumărate ori în contexte diferite"

- Design Patterns: Elements of Reusable Object-Oriented Software – 1994

- Gang of Four (GoF)- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

- Introduce șabloanele de proectare și oferă un catalog de șabloane

Tipuri de șabloane de proiectare (dupa scop):

- **Creaționale**
  - descriu modul de creare a obiectelor
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton

- **Structurale**
  - se refere la compoziția claselor sau al obiectelor
  - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

- **Comportamentale**
  - descriu modul în care obiectele și clasele interacționează și modul în care distribuim responsabilitățile
  - Chain of responsibility, Command Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

Elemente de descriu un șablon de proiectare

- Numele șablonului
  - descrie sintetic problema rezolvată și soluția
  - face parte din vocabularul programatorului

- Problema
  - Descrie problema și contextul în care putem aplica șablonul.

- Soluția
  - Descrie elementele soluției, relațiile între ele, responsabilitățile și modul de colaburare
  - Oferă o descriere abstractă a problemei de rezolvat, descrie modul de aranjare a elementelor (clase, obiecte) din soluție

- Consecințe
  - descrie consecințe, compromisuri legat de aplicarea șablonului de proiectare.

Standard Template Library (STL)

- The Standard Template Library (STL), este o bibliotechă de clase C++, parte din C++ Standard Library
- Oferă structuri de date și algoritmi fundamentali, folosiți la dezvoltarea de programe in C++
- STL oferă componente generice, parametrizabile. Aprope toate clasele din STL sunt parametrizate (Template).
- STL a fost conceput astfel încât componentele STL se pot compune cu usurință fără a sacrifica performanța (generic programming)
- STL conține clase pentru:
  - containere
  - iteratori
  - algoritmi
  - function objects
  - allocators

Containers

Un container este o grupare de date în care se pot adauga (insera) si din care se pot sterge (extrage) obiecte. Implementările din STL folosesc șabloane ceea ce oferă o flexibilitate în ceea ce privește tipurile de date ce sunt suportate

Containerul gestionează memoria necesară stocarii elementelor, oferă metode de acces la elemente (direct si prin iteratori)

Toate containerele oferă funcționalități (metode):

- accesare elemente (ex.: [])
- gestiune capacitate (ex.: size())
- modificare elemente (ex.: insert, clear)
- iterator (begin(), end())
- alte operații (ie: find)

Decizia în alegerea containerului potrivit pentru o problemă concretă se bazează pe:

- funcționalitățile oferite de container
- eficiența operațiilor (complexitate).

Containere - Clase template

- Container de tip secvență (Sequence containers): **vector<T>**, **deque<T>**, **list<T>**
- Adaptor de containere (Container adaptors): **stack<T, ContainerT>**, **queue<T, ContainerT>**, **priority_queue<T, ContainerT, CompareT>**
- Container asociativ (Associative containers): **set<T, CompareT>**, **multiset<T, CompareT>**, **map<KeyT, ValueT, CompareT>**, **multimap<KeyT, ValueT, CompareT>**, **bitset<T>**

Container de tip secvență (Sequence containers):

Vector, Deque, List sunte containere de tip secvență, folosesc reprezentări interne diferite, astfel operațiile uzuale au complexități diferite

- Vector (Dynamic Array):
  - elementele sunt stocate secvențial in zone continue de memorie
  - Vector are performanțe bune la:
    - Acesare elemente individuale de pe o poziție dată(constant time).
    - Iterare elemente în orice ordine (linear time).
    - Adăugare/Ștergere elemente de la sfârțit(constant amortized time).

- Deque (double ended queue) - Coadă dublă (completă)
  - elementele sunt stocate în blocuri de memorie (chunks of storage)
  - Elementele se pot adăuga/șterge eficient de la ambele capete

- List
  - implementat ca și listă dublă înlănțuită
  - List are performanțe bune la:
    - Ștergere/adăugare de elemente pe orice poziție (constant time).
    - Mutarea de elemente sau secvențe de elemente în liste sau chiar și intre liste diferite (constant time).
    - Iterare de elemente in ordine (linear time).

Operații / complexity

```cpp
#include <vector>
void sampleVector() {
 vector<int> v;
 v.push_back(4);
 v.push_back(8);
 v.push_back(12);
 v[2] = v[0] + 2;
 int lg = v.size();
 for (int i = 0; i < lg; i++)
 {
   cout << v.at(i) << " ";
 }
}
```

```cpp
#include <deque>
void sampleDeque() {
 deque<double> dq;
 dq.push_back(4);
 dq.push_back(8);
 dq.push_back(12);
 dq[2] = dq[0] + 2;
 int lg = dq.size();
 for (int i = 0; i < lg; i++)
 {
   cout << dq.at(i) << " ";
  }
}
```

```cpp
#include <list>
void sampleList() {
 list<double> l;
 l.push_back(4);
 l.push_back(8);
 l.push_back(12);
 while (!l.empty()) {
   cout << " " << l.front();
   l.pop_front();
 }
}
```

Vector : timp constant O(1) random access; insert/delete de la sfârșit

Deque: timp constant O(1) insert/delete at the either end

List: timp constant O(1) insert / delete oriunde în listă

Vector vs Deque

- Accesul la elemente de pe orice poziție este mai eficient la vector
- Inserare/ștergerea elementelor de pe orice poziție este mai eficient la Deque (dar nu e timp constant)
- Pentru liste mari Vector alocă zone mari de memorie, deque aloca multe zone mai mici de memorie – Deque este mai eficient în gestiunea memoriei

Warehouse – folosind vector

```
/**
 * Store the products in memory (in a dynamic array)
 */
class ProductInMemoryRepository: public ProductRepository {
public:
	/**
	 * Store a product
	 * p - product to be stored
	 * throw RepositoryException if a product with the same id already exists
	 */
	void store(Product& p) throw (RepositoryException);
	/**
	 * Lookup product by code
	 * the code of the product
	 * return the product with the given code or NULL if the product not found
	 */
	const Product* getByCode(int code);
	/**
	 * Count the number of products in the repository
	 */
	int getNrProducts();

	ProductInMemoryRepository();
	~ProductInMemoryRepository();
private:
	vector<Product*> prods;
};



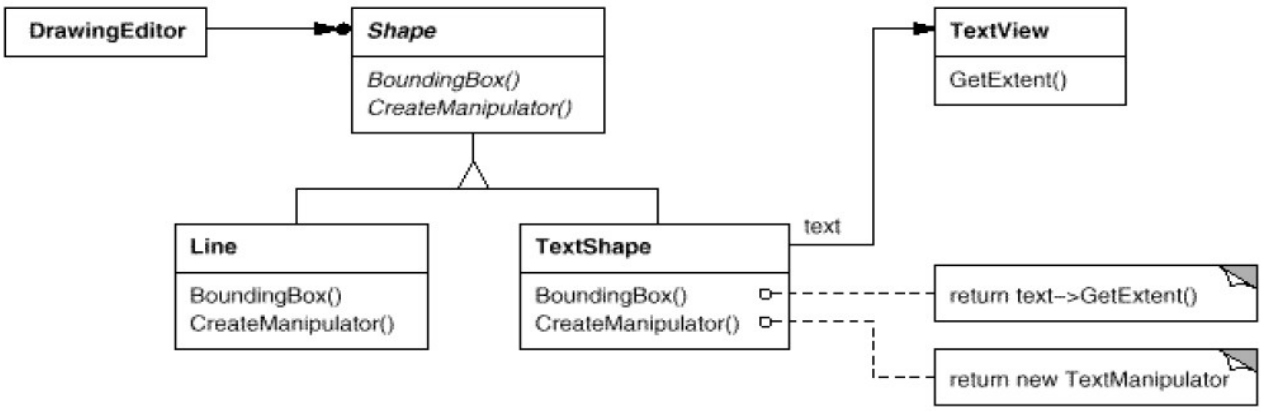
/**
 * Store a product
 * p - product to be stored
 * throw RepositoryException if a product with the same id already exists
 */
void ProductInMemoryRepository::store(Product& p) throw (RepositoryException) {
	//verify if we have a product withe same code
	const Product* aux = getByCode(p.getCode());
	if (aux != NULL) {
		throw RepositoryException("Product with the same code already exists");
	}
	prods.push_back(&p);
}
```

Adapter pattern (Wrapper)

Intenția: Adaptarea interfeței unei clase la o interfață potrivită pentru client. Permite claselor sa interopereze care fara convertirea interfeței nu ar putea conlucra.

Motivație: În unele cazuri avem clase din biblioteci externe care ar fi potrivite ca și funcționalitate dar nu le putem folosi pentru ca este nevoie de o interfață specifică în codul existent în aplicație.

Ex. Draw Editor (Shape: lines, polygons, etc) Add TextShape. Soluția este sa adaptăm clasa existentă TextView class. TexShape adaptează clasa TextView la interfața Shape

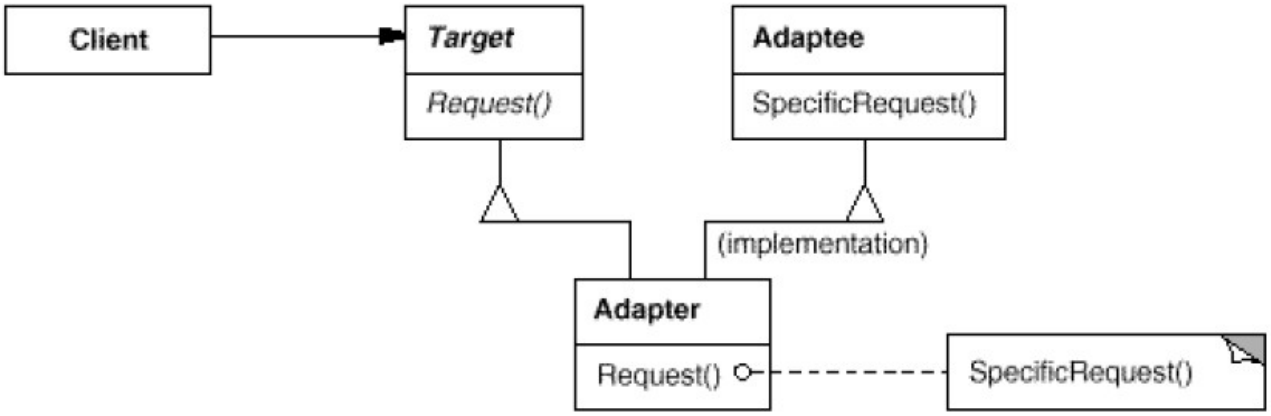


Aplicabilitate:

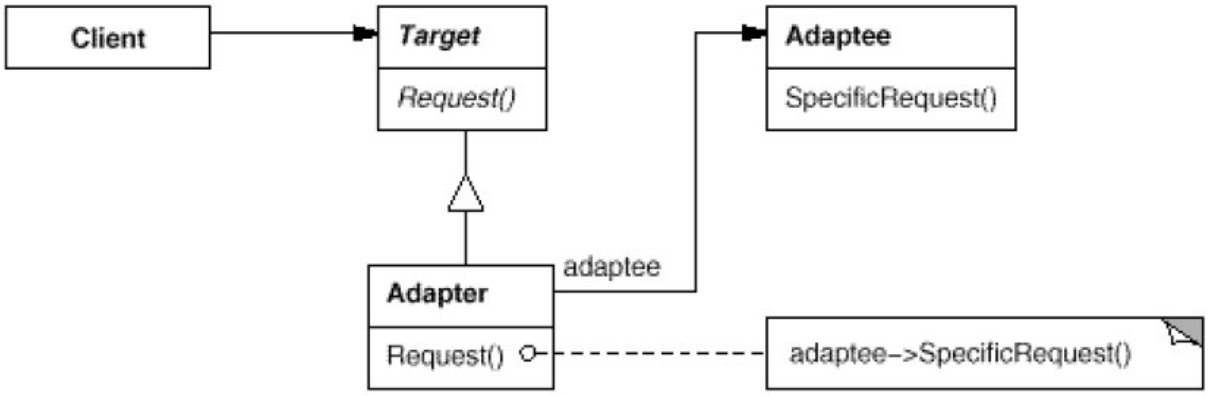
- dorim să folosim o clasă existentă dar interfața clasei nu corespunde cu ceea ce este nevoie
- creare de clase reutilizabile care cooperează cu alte clase (dar ele nu au interfețe compatibile)

Adapter - structură

Class adapter – folosește moștenire multiplă



Object adapter folosește compoziție



Participants:

- Target: definește interfața de care este nevoie.
- Client: colaborează, folosește obiecte cu interfață Target.
- Adaptee: este clasa care trebuie adaptată. Are interfața diferită de ceea ce e are nevoie Client
- Adapter: adaptează Adaptee la interfața Target.

Adapter

Colaborare:

- Clientul apelează metode al lui Adapter. Clasa adapter folosțe metode de la clasa Adaptee pentru a efectua operația dorită de Client.

Consecințe:

Class adapter:

- Nu putem folosi dacă dorim sa adaptăm clasa și toate clasele derivate
- Permite clasei Adapter să suprascrie anumite metode a clasei Adaptee
- Introduce un singur obiect nou în sistem. Metodele din adapter apelează direct metode din Adaptee

Object adapter:

- este posibil ca un singur Adapter sa foloseasca mai multe obiecte Adaptees.
- Este mai dificil să suprascriem metode din Adaptee (Trebuie sa creăm o clasa derivată din Adaptee și sa folosim această clasă derivată în clasa Adapter)

Adapter folosit în STL: Container adapters, Iterator adapters

Adaptor de containere (Container adaptors)

Sunt containere care încapsulează un container de tip secvență, și folosesc acest obiect pentru a oferi funcționalitățisi specifice containerului (stivă, coadă, coadă cu priorități).

STL folosește șablonul adapter pentru: Stack, Queue, Priority Queue. Aceste clase au un template parameter de tip container de secvență, dar oferă doar operații permise pe stivă, coadă, coadă cu priorități (Stack, Queue, Priority Queue)

- Stack: strategia LIFO (last in first out) pentru adaugare/ștergere elemente
  - Elemente sunt adăugate/extrase la un capăt (din vârful stivei)
  - Operații: empty(), push(), pop(), top()
  - ```
    template < class T, class Container = deque<T> > class stack;
    ```
    - T: tipul elementelor
    - Container: tipul containerului folosit pentru a stoca elementele din stivă

- queue: strategia FIFO (first in first out)
  - Elementele sunt adăugate (pushed) la un capăt și extrase (popped) din capătul celălalt
  - operații: empty(), front(), back(), push(), pop(), size();
  - ```
    template < class T, class Container = deque<T> > class queue;
    ```

- priority_queue: se extrag elemente pe baza priorităților
  - operations: empty(), top(), push(), pop(), size();
  - ```
    template < class T, class Container = vector<T>,
        class Compare = less<typename Container::value_type> >
    class priority_queue;
    ```
    ```

Adaptor de containere - exemple

```
#include <stack>
void sampleStack() {
 stack<int> s;
 //stack<int,deque<int>> s;
 //stack<int,list<int> > s;
//stack<int,vector<int>> s;
 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty()) {
    cout << s.top() << " ";
    s.pop();
 }
}
```

```
#include <queue>
void sampleQueue() {
//queue<int> s;
//queue<int,deque<int>>s;
queue<int, list<int> > s;

 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty()) {
  cout << s.front() << "
";
  s.pop();
 }
}
```

```
#include <queue>
void samplePriorQueue() {
//priority_queue<int> s;
//priority_queue<int,deque<int>> s;
//priority_queue<int,list<int>> s;

priority_queue<int,vector<int> > s;
 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty()) {
   cout << s.top() << " ";
   s.pop();
 }
}
```

Container asociativ (Associative containers):

Sunt eficiente în accesare elementelor folosind chei (nu folosind poziții ca și în cazul containerelor de tip secvență).

- set
  - mulțime - stochează elemente distincte. Elementele sunt folosite și ca și cheie
  - nu putem avea doua elemente care sunt egale
  - se folosește arbore binar de căutare ca și reprezentare internă

- Map
  - dictionar - stochează elemente formate din cheie și valoare
  - nu putem avea chei duplicate

- Bitset
  - container special pentru a stoca biti (elemente cu doar 2 valori posibile: 0 sau 1,true sau false, ...).

```
void sampleMap() {
	map<int, Product*> m;
	Product *p = new Product(1, "asdas", 2.3);
	//add code <=> product
	m.insert(pair<int, Product*>(p->getCode(), p));

	Product *p2 = new Product(2, "b", 2.3);
	//add code <=> product
	m[p2->getCode()] = p2;

    //lookup
	cout << m.find(1)->second->getName()<<endl;
	cout << m.find(2)->second->getName()<<endl;
}
```

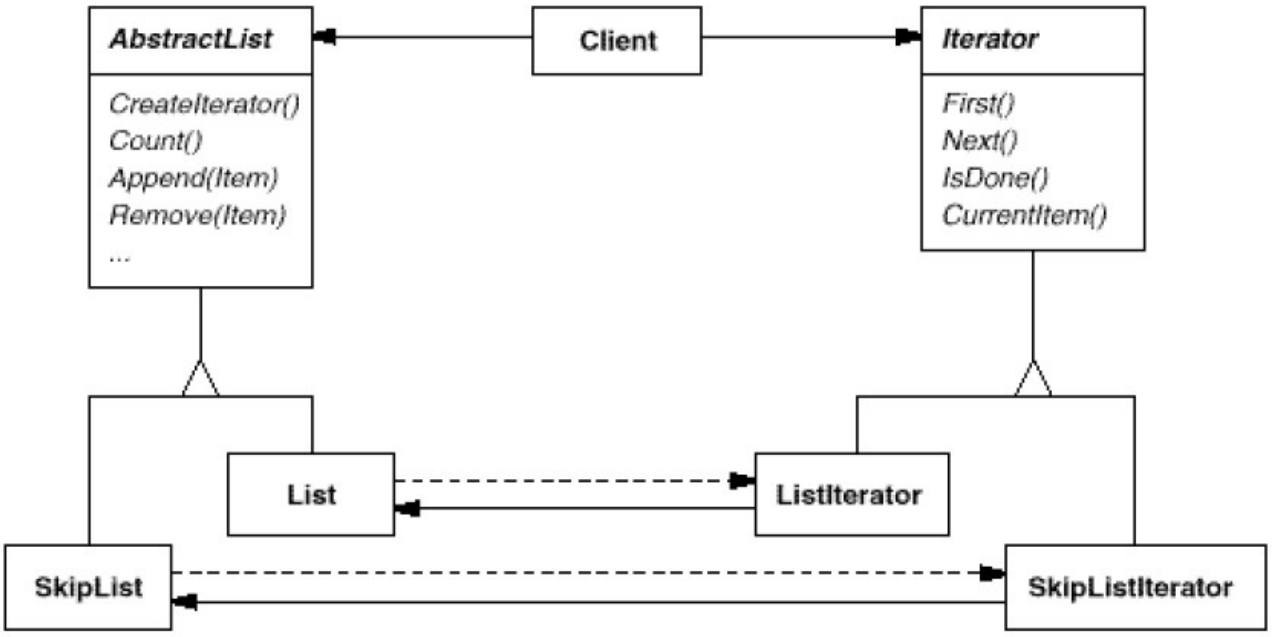
Şablonul Iterator (Cursor)

Intenţie: Oferă acces secvențial la elementele unui agregat fără a expune reprezentarea internă.

Motivație:

- Un agregate (ex. listă) ar trebui sa permite accessul la elemente fără a expune reprezentarea internă
- Ar trebui sa permite traversarea elementelor în moduri diferite (inainte, înapoi, random)

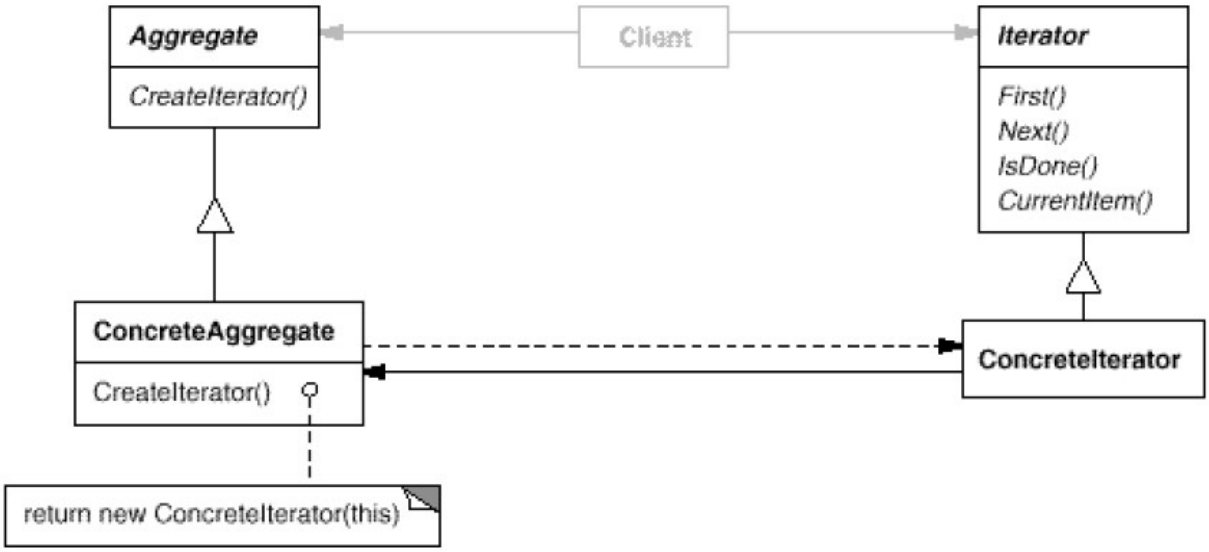
Exemplu: List, SkipList, Iterator



Aplicabilitate:

- diferite tipuri de traversari pentru un agregat
- oferă o interfață uniformă pentru accesul la elementele unui agregat

Iterator design pattern structure



Participants:

Iterator: defineşte interfaţa pentru a traversa elementele

ConcreteIterator: Implementează interfaţă Iterator, responsabil cu gestiunea poziţiei curente din iteraţie

Aggregate: defineşte metode pentru a crea un obiect de tip iterator

ConcreteAggregate: Implementează interfaţa necesară pentru creare de Iterator crează ConcreteIterator

Consecinţe:

- suportă multiple tipuri de traversari. Agregate mai complexe au nevoie de diferite metode prin care se accesează elementele
- se simplifică interfaţa Aggregate.
- Putem avea mai multe traversari în acelaşi moment.

Iteratori in STL

Iterator: obiect care gestionează o poziție (curentă) din containerul asociat. Oferă suport pentru traversare (++,--), dereferențiere (*it).

Iteratorul este un concept fundamental in STL, este elementul central pentru algoritmi oferiţi de STL.

Fiecare container STL include funcţii membre begin() and end()

```
void sampleIterator() {
	vector<int> v;
	v.push_back(4);
	v.push_back(8);
	v.push_back(12);
	//Obtain an the start of the iteration
	vector<int>::iterator it = v.begin();
	while (it != v.end()) {
		//dereference
		cout << (*it) << " ";
		//go to the next element
		it++;
	}
	cout << endl;
}
```

Permite decuplarea intre algoritmi si containere

Existe mai multe tipuri de iteratori:

- iterator input/output (istream_iterator, ostream_iterator)
- forward iterators, iterator bidirectional, iterator random access
- reverse iterators