Generic design patterns

Bo Simonsen
bosim@diku.dk

Department of Computing
The University of Copenhagen

The 4th of February , 2009

Generic programming

- What is generic programming?

Patterns

- Idioms
- Pattern overview
- Generic bridge pattern
- Generic decorator pattern
- Generic adapter pattern
- Generic proxy pattern
- Generic strategy pattern
- Generic visitor pattern
- Generic iterator pattern
- Generic template method

Design in the CPH STL

What generic programming?

- Programming with templates
- Template parameters are substituted on compile-time to actual instances.

The STL

- Consists of Containers, Iterators and Algorithms.
- Containers: list, vector, deque, set, multiset, map, and multimap.
- Algorithms: `for_each`, `next_permutation`, ...
- An important design principle: Value semantics.

Pattern hierarchy

The hierarchy of patterns (top-down):

- Software architectural patterns (layers, client-server).
- Design patterns.
- Idioms.

Generic idioms

- A idiom: *An idiom is a phrase whose meaning cannot be determined by the literal definition of the phrase itself, but refers instead to a figurative meaning that is known only through common use.*
- A low-level pattern
- For example, the CRTP idiom: Give the inheriting class as a template argument to the base class. This is usefull when explicit instantiation is performed in the base class, which should be of the inheriting class.

CRTP example

A base class for node which is used in a tree structure can be defined to be 3 pointers, left, right and parent. A node for e.g. the AVL tree inherits from this class, to ensure that the pointer is of the AVL-tree node type, we can use CRTP.

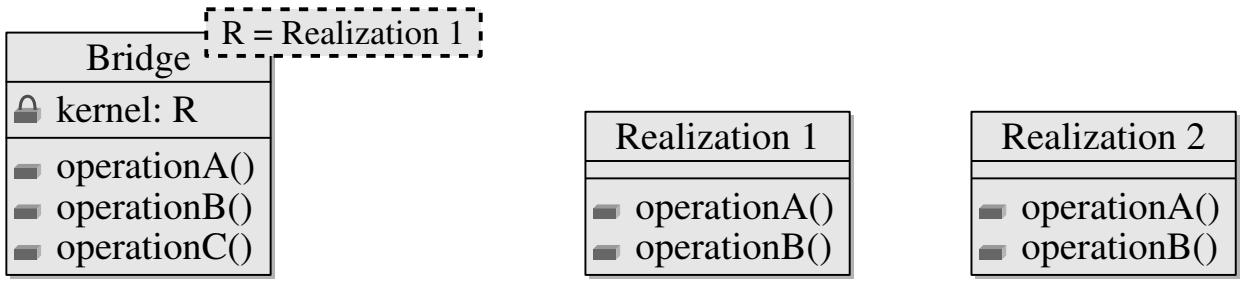
```
template <typename N> class node_base {
private:
  N *left, *right, *parent;
public:
  /* Methods for accessing pointers */
};

template <typename V>
class avl_node : public node_base< avl_node<V> > {
  ...
};
```

Generic design patterns

- Structural
  - Bridge
  - Decorator
  - Adaptor
  - Proxy
- Behavioural
  - Strategy
  - Iterator

Generic bridge pattern

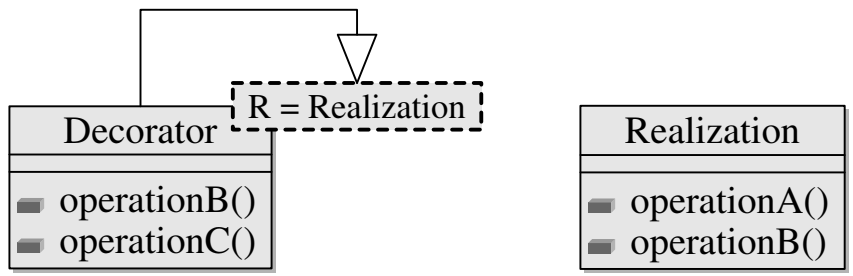
Intention: Decouple an abstraction from its implementation so that the two can vary independently.



CPH STL Usage: Usage: In the CPH STL we have several data structures which implements the same container. For example, AVL and red-black trees both implements set, map, multiset and multimap.
Advantages: Some of the code can be implemented entirely within the bridge classes.

Generic decorator pattern

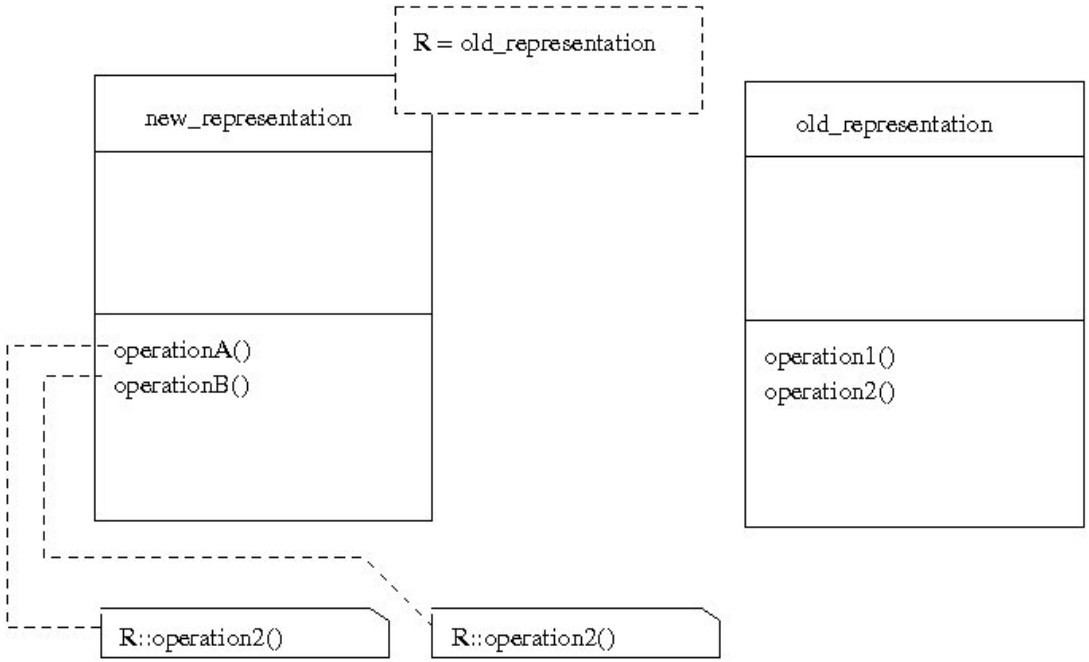
Intention: Define additional responsibilities to a set of objects or replace functionalities of a set of objects.
Definition: Decorators provide a flexible alternative to subclassing for extending functionality.



CPH STL Usage: When creating smart iterators, for example, a functor iterator (instead of x we return $f(x)$) the decorator should be used.
Advantages: If we should use inheritance we could not let the implementation vary independently.

Generic adapter pattern

Intention: Convert the interface of a class into another interface expected by specific clients. Adapter lets classes work together that could not otherwise be possible because of incompatible interfaces.



Generic adapter pattern (continued)

STL usage: Adapters are used, for example, in the `stack` container. `stack` is given a container, where the data is stored, the adaptor is simply translating between the stack interface and the container interface. The operations for stack `s` and container `c` are:

- `s.top()`: `c.front()`
- `s.pop()`: `c.pop_front()`
- `s.push()` : `c.push_front()`

Generic proxy pattern

Intention: Provide a surrogate or placeholder for another object to control access to it.
Structure: Similar to bridge, but with some additional functionality. The additional functionality could be to implement lazy retrival of files, so first when the file is used it is opened.
CPH STL problem: Regarding exception safety, `swap` must not throw an exception, but when two containers are swaped, an exception can occur during copy construction of allocator and comparator.

Generic proxy pattern (continued)

Solution: Keep the comparator and allocator as pointers. For this purpose we have created a `comparator_proxy` and `allocator_proxy`. Allocator and comparator operations are now transparant.

```
template <typename C> class comparator_proxy {
public:
  /* Constructors are ommited, they allocate the real comparat
  bool operator()(const first_argument_type& t1,
                  const second_argument_type& t2) {
    return (*(*this).c)(t1, t2);
  }
  void swap(comparator_proxy& o) {
    std::swap(o.c, (*this).c);
  }
private:
  C* c;
};
```

Generic proxy pattern (continued)

Advantages: It is possible to add additional functionality without making significant changes to the excisting code. In our example, the original code from the container was:

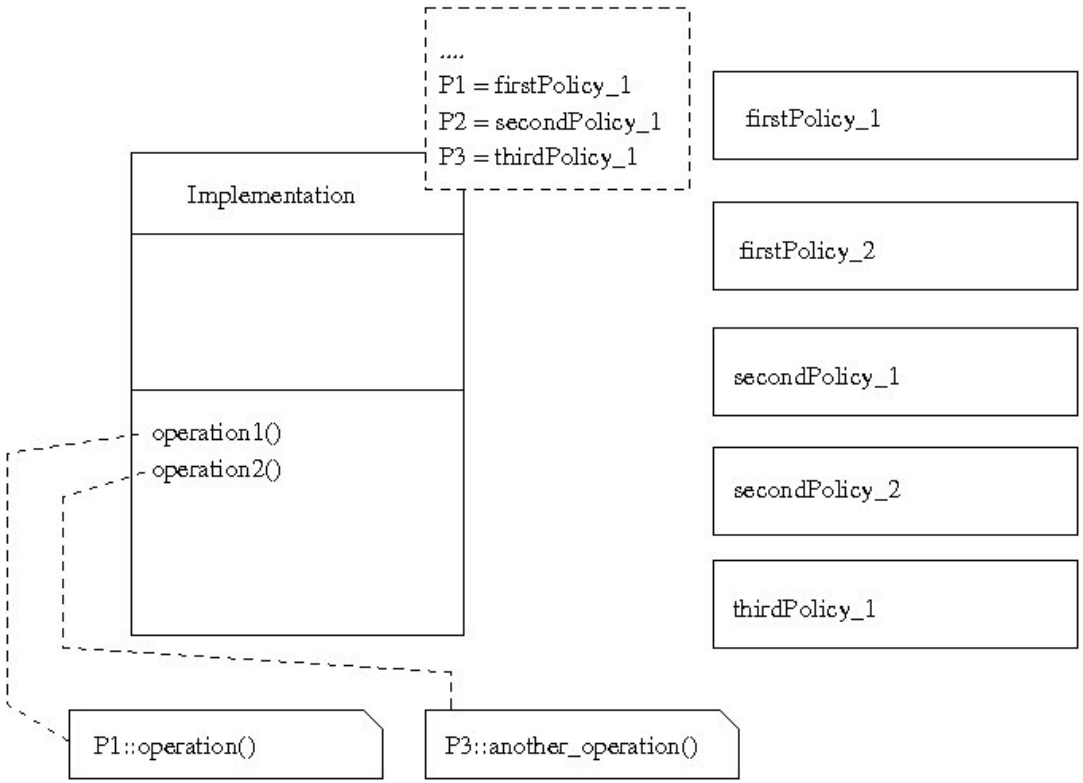
```
template <..., typename C, ...>
class tree {
public:
  typedef C comparator_type;
  ..
private:
  comparator_type comparator;
};
```

Now:

```
template <..., typename C, ...>
class tree {
public:
  typedef comparator_proxy<C> comparator_type;
...
};
```

Generic strategy pattern

Intention: Define a family of strategies, encapsulate each one, and make them interchangeable.



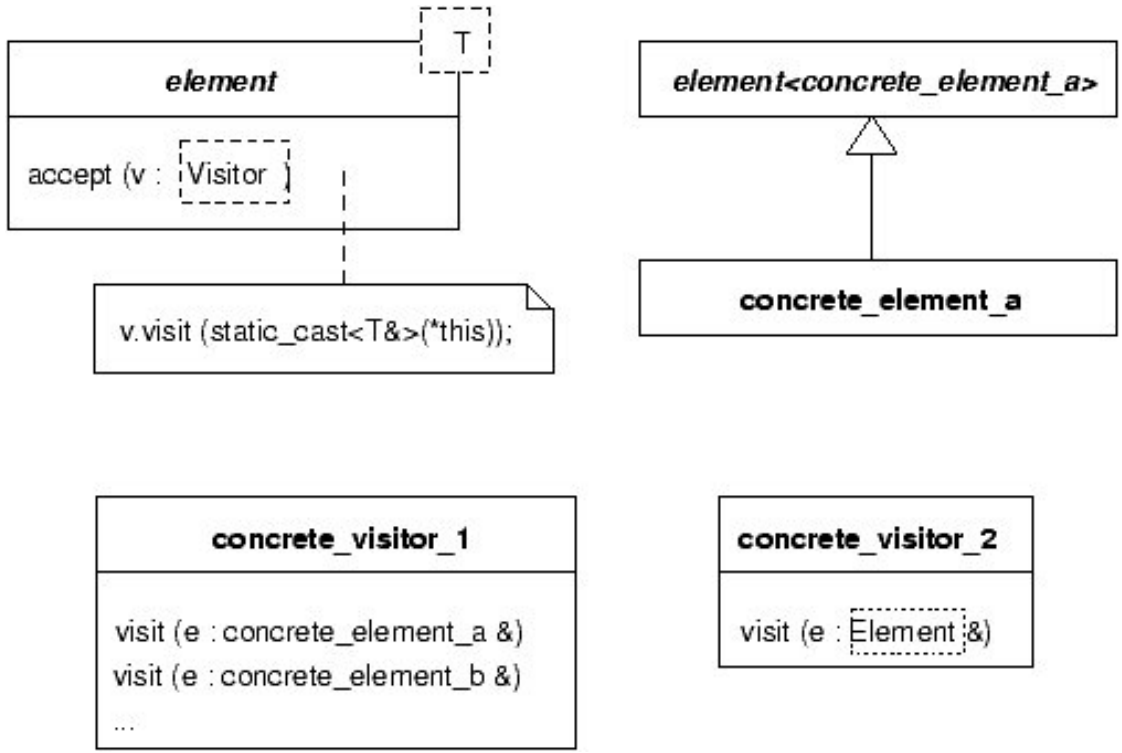
In the context of generic programming strategies is usually denoted *policies*.

Generic strategy pattern (continued)

STL example: The comparator and allocator is given to containers as policies. For ordered containers, the user can now define the ordering. The classic example is: If a queue is given `std::less` as comparator the priority queue will be max ordered, if it is given `std::greater` it will be min ordered.
CPH STL example: In addition, all containers in the CPH STL accepts a storage policy. This policy defined how data should be stored, this makes for example, bit-packing possible without changing the code in the container.

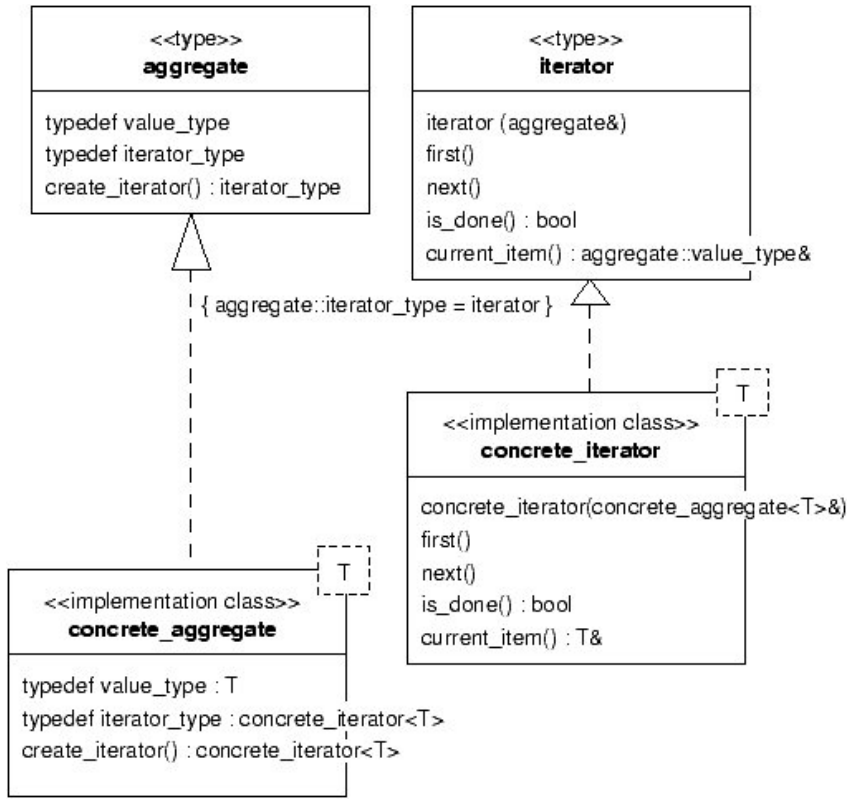
Generic visitor pattern

Intention: To define a new operation for the concrete classes of a hierarchy without modifying the hierarchy.



Generic iterator pattern

Intention: Provide a way to access the elements of an aggregate object without exposing its underlying representation.

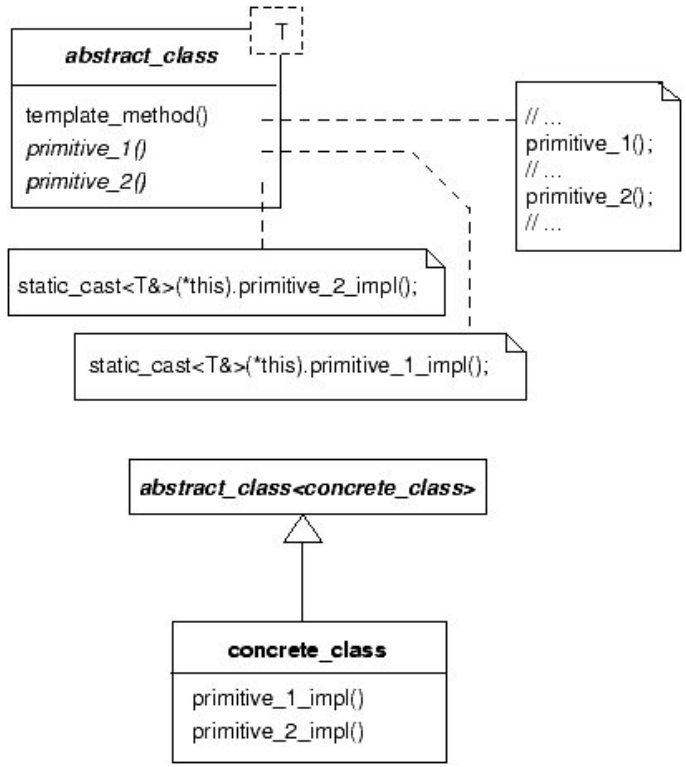


Generic iterator pattern (continued)

CPH STL usage: Iterators are a fundemental part of the STL, which makes it possible for the algorithms to access container data.
Advantages: A unified way for accessing data.

Generic template method

Intention: To define the canvas of an efficient algorithm in a superior class, deferring some steps to subclasses.



Design in the CPH STL

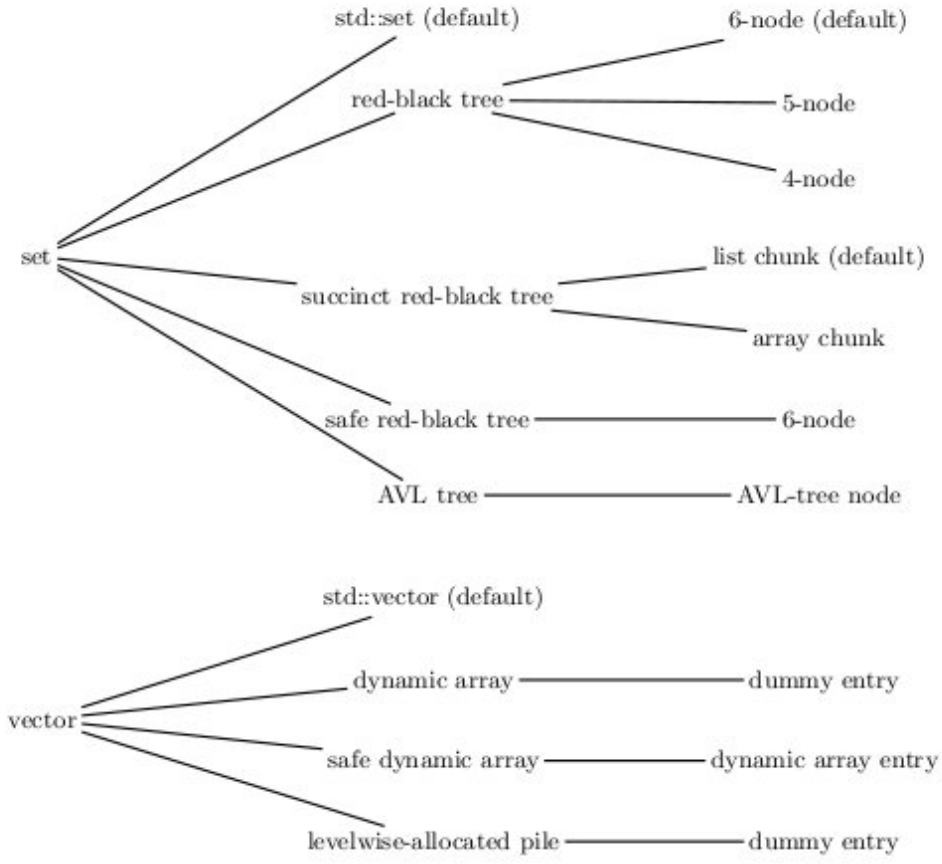
Requirements:

- Several data structures implements the same container.
- Maximize code reuse, for better mentainability.
- Provide safe iterators, the user should not be able to damage the internal structure using the iterator.

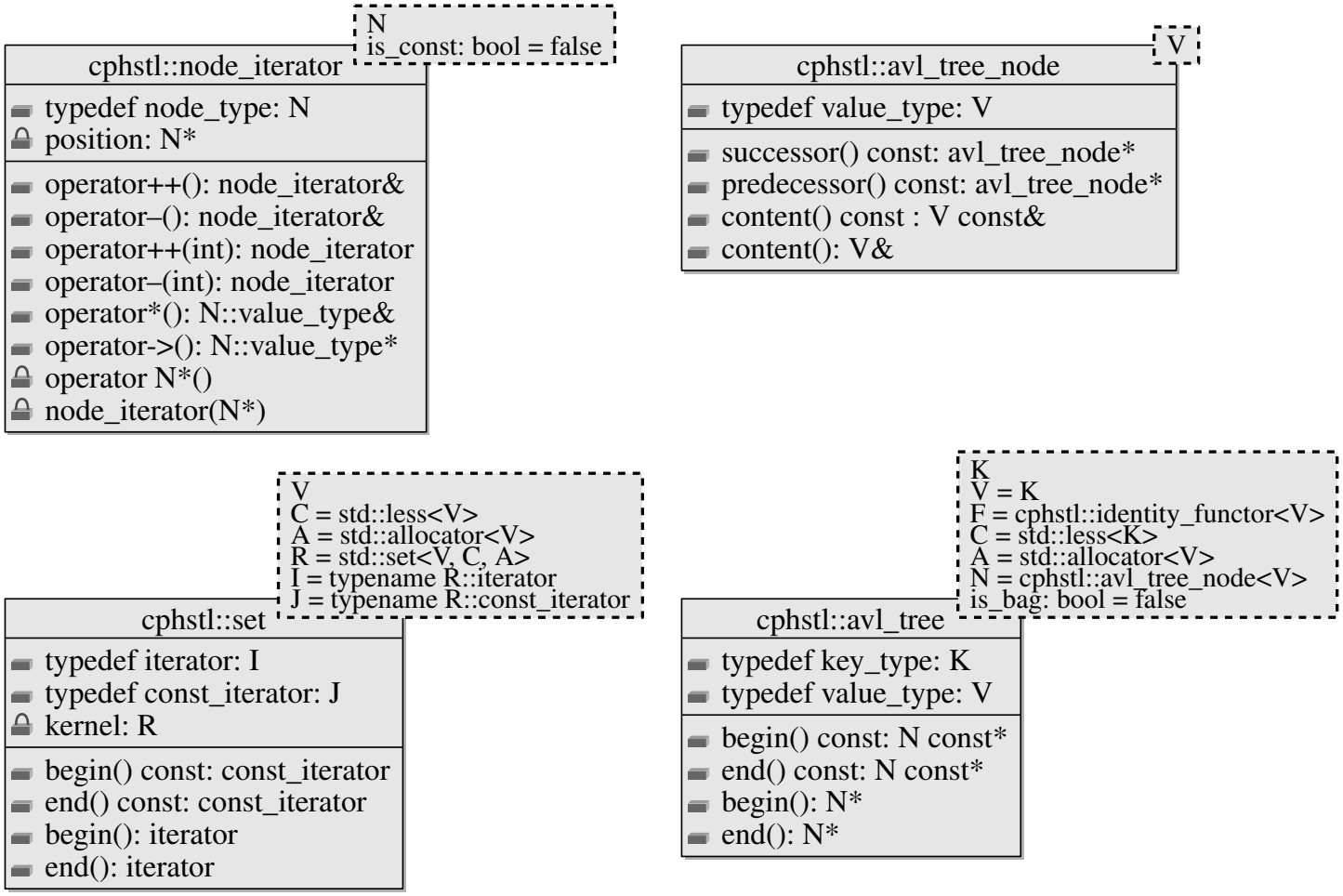
Terminology

- *container* - Often the bridge class, the class which is available to the user.
- *realization* - The class which realizeses the container. An implmenentation of the data structure
- *abstract iterator* - The iterator which is given to the user from the container.
- *concrete iterator* - The iterator which is given to the bridge from the realization.
- *storage policy* - Defines a containers internal structure.

Containers, Realizations, and Storage Policies



Relationship between components



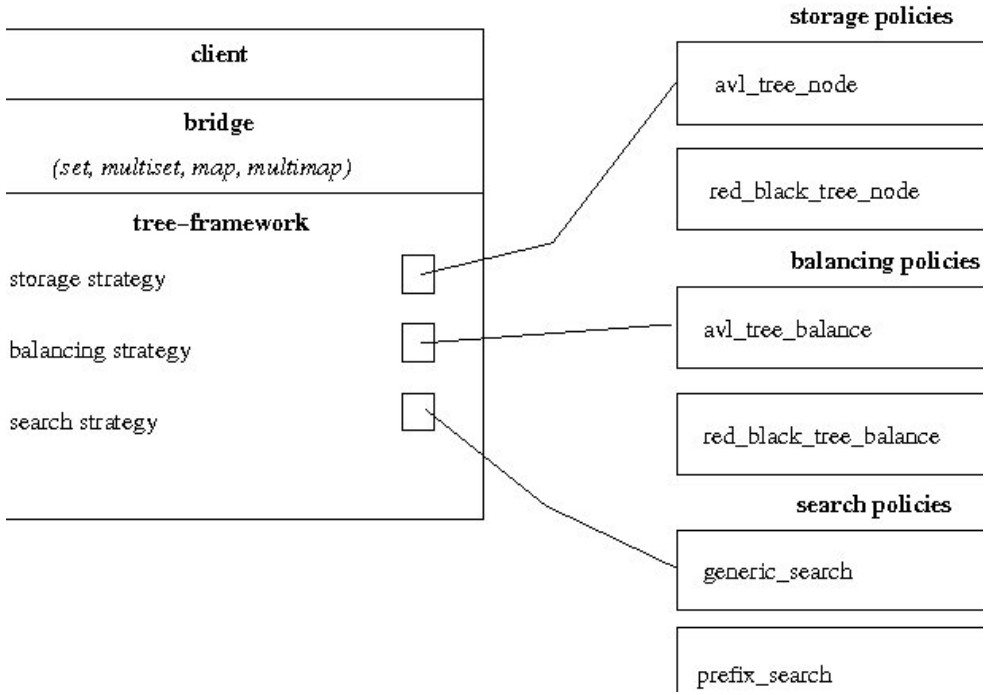
Key points

- The behaviour of the iterator is defined using the methods in the node class, the node iterator is just an interface. All containers based on nodes can share the same interface. This implies code reuse.
- Convertion between concrete and abstract iterators is done transparantly. abstract iterator $\longrightarrow$ concrete iterator: convertion operator, concrete iterator $\longrightarrow$ abstract iterator: parameterized constructor.
- Encapsulation is done using friend declarations. Only bridge classes can access the private members of the abstract iterator.

Exercise

`insert(value_type const&)` in set returns a pair of a iterator and a boolean value (`std::pair<iterator, bool>`). The iterator points to the newly inserted value or to the existing value if the element was already existing. The boolean value will be true if the element was inserted, or false if the element was already existing. Will convertion between abstract iterators and concrete work here? And why?

More policy based design



Litterature

- Katajainen, J., Simonsen B.: Applying Design Patterns to Specify the Architecture of a Generic Program Library, Not published yet (2008)
- Duret-Lutz, A., Géraud, T., Demaille, A.: Design patterns for generic programming in C++. In: Proceedings of the 6th Conference on USENIX Conference on Object—Oriented Technologies and Systems, The USENIX Association (2001) 189–202
- Simonsen B.: Refactoring the CPH STL: Designing an independent and generic iterator, CPH STL Report 2008-6, Department of Computing, University of Copenhagen (2008)