Selected topics in software development

Today:

Design patterns used in the CPH STL

Speaker:

Jyrki Katajainen

Course home page:

`http://www.diku.dk/forskning/performance-engineering/Software-development/`

Adapter pattern

Convert the interface of a class into another that clients expect. Adapter lets classes work together that would not be otherwise possible because of incompatible interfaces.

- Old structures can be used in a new context.

Stack as an adapter

```
template <
  typename E,
  typename R = cphstl::deque<E>
>
class stack {
public:
  typedef E value_type;
  typedef std::size_t size_type;
  typedef R container_type;
protected:
  R sequence;
public:
  explicit stack(R const& = R());
  bool empty() const;
  size_type size() const;
  E& top();
  E const& top() const;
  void push(E const&);
  void pop();
};
```

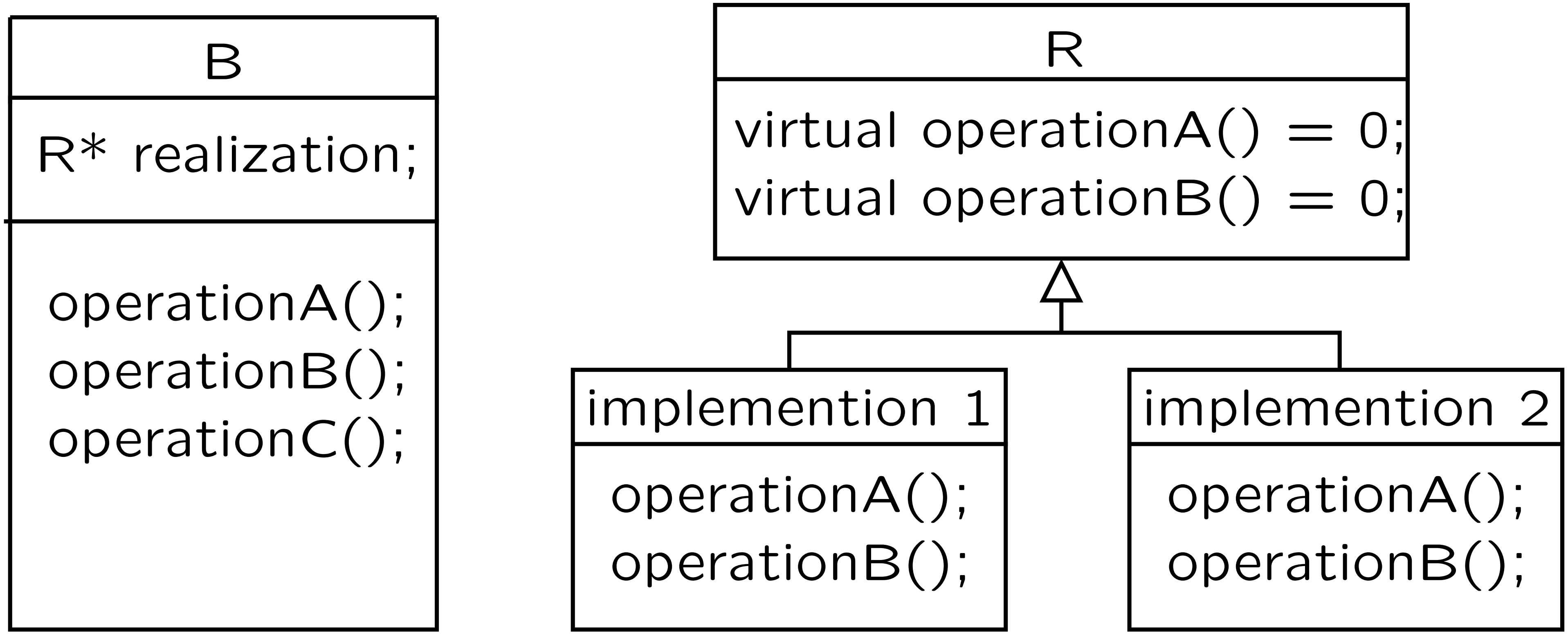
```
template <
  typename E,
  typename A = std::allocator<E>
>
class deque {
public:
  ...
  typedef E& reference;
  typedef E const& const_reference;
  ...
  E& front();
  E const& front() const;
  E& back();
  E const& back() const;
  ...
  void push_front(E const&);
  void push_back(E const&);
  void pop_front();
  void pop_back();
  ...
};
```

Bridge pattern

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

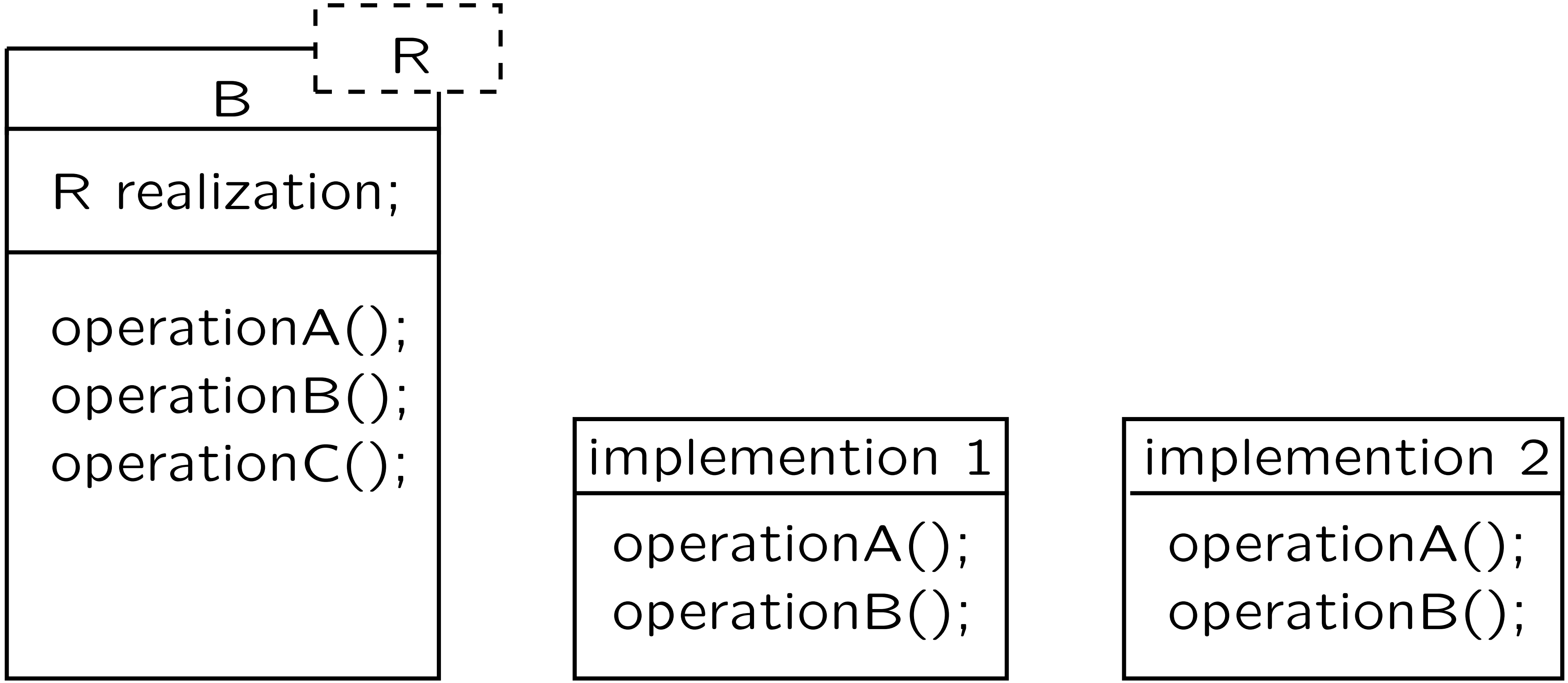
- Possible to provide several implementations with the same interface.
- Clients can select the best implementations for their purposes.
- Implementations can be smaller than the bridge (that is, pieces identical to all implementations are implemented at the bridge).

Bridge pattern implemented using inheritance



Source: [Vandevoorde and Josuttis 2003, §14.4]

Bridge pattern implemented using templates



Source: [Vandevoorde and Josuttis 2003, §14.4]

Stack as a container

```
template <
  typename E,
  typename A = std::allocator<E>,
  typename R = cphstl::list_stack<E, A>
>
class stack {
public:
  ...
  size_type size() const;
  bool empty() const;
protected:
  R container;
};
```

```
template <
  typename E,
  typename A = std::allocator<E>,
  typename R = std::list<E, A>
>
class list_stack {
public:
  ...
  typedef std::size_t size_type;
  ...
  size_type size() const;
  ...
};
```

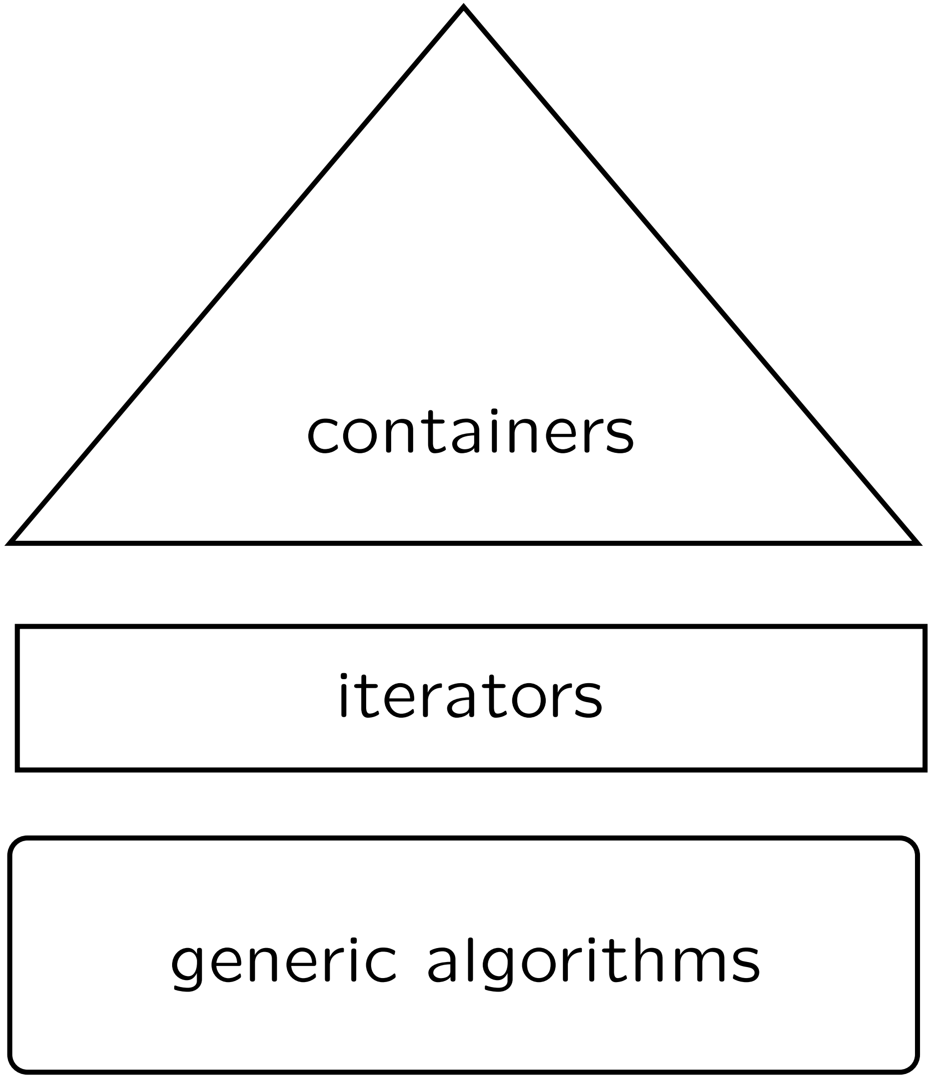
```
template <typename E, typename A, typename R>
typename stack<E, A, R>::size_type
stack<E, A, R>::size() const {
  return container.size();
}
template <typename E, typename A, typename R>
bool
stack<E, A, R>::empty() const {
  return (*this).size() == 0;
}
```

Iterator pattern

Provide a way to access the elements of a container sequentially without exposing its underlying representation.

- In the STL, iterators come in several different flavours: locators (or trivial iterators), input iterators, output iterators, forward iterators, bidirectional iterators, and random-access iterators.
- Iterators are generalizations of pointers.

Iterators as the clue



Generic merge routine

```
#include <list>
#include <deque>
#include <algorithm>
#include <cassert>

template <typename sequence>
sequence make(char const s[]) {
  return sequence(&s[0], &s[std::strlen(s)]);
}

int main () {
  char* vowels = "aeiouy";
  int len = std::strlen(vowels);

  std::list<char> consonants = make<std::list<char> >("bcdfghjklmnpqrstvwxz");

  std::deque<char> alphabet(26, ' ');

  std::merge(&vowels[0], &vowels[len], consonants.begin(), consonants.end(),
             alphabet.begin());

  assert(alphabet == make<std::deque<char> >("abcdefghijklmnopqrstuvwxyz"));
  return 0;
}
```

Strategy pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- Again there are two variations depending on whether the pattern is used at compile time or at run time.
- Clients can select the best algorithm for their purposes.

Sorting

```
template <
  typename I, // random access iterator
  typename C, // comparator
  typename R = cphstl::mergesort(I, I, C)
>
void
sort(I, I, C, R = R());
```

```
template <
  typename I,
  typename C = std::less<E>
>
class mergesort {
public:
  operator()(I, I, C = C());
};
template <
  typename I,
  typename C = std::less<E>
>
class quicksort {
public:
  operator()(I, I, C = C());
};
template <
  typename I,
  typename C = std::less<E>
>
class heapsort {
public:
  operator()(I, I, C = C());
};
```

Proxy pattern

Provide a surrogate or placeholder for another object to control access to it.

References to bits

```
class reference {
private:
  bitset<N, word_t>& bs;
  size_t pos;

  friend class bitset;

  reference();
  reference(bitset<N, word_t>&, size_t);
public:
  ~reference()
  reference& operator=(bool x); // for b[i] = x;
  reference& operator=(reference const&); // for b[i] = b[j];
  bool operator~() const; // flips the bit
  operator bool() const; // for x = b[i];
  reference& flip(); // for b[i].flip();
};
```

Factory-method pattern

Define an interface for creating an object, but let subclasses decide which class to instantiate. factory method lets a class defer instantiation to subclasses.

- Again there are two incarnations depending on whether we rely on inheritance (subclasses) or templates (partial specialization).

Factory of universal hash functions

```
template <
  typename D,
  typename R
>
class universal_hash_function {
public:
  typedef D domain_type;
  typedef R range_type;
};

template <
  typename integer,
  typename R = std::size_t
>
class universal_hash_function<integer, R> {
public:
  typedef integer domain_type;
  typedef R range_type;

  universal_hash_function();
  R evaluate(integer);
  void reconstruct();
private:
  ...
};
```

Conclusion

- Continue reading until you understand the short descriptions of the design patterns given on the cover of the pattern-catalogue book.