CS 36300 / ECE 49500
PRINCIPLES OF SOFTWARE DESIGN

050. Software Design Patterns

The (Wrapper) Façade Pattern

Have you ever been in the following situation:

- Dealing with many interacting objects (or subsystems) to achieve some higher-level functionality
- Working with low-level functionality and APIs, and need to logically coordinate invocations
- Need to shield developers from low-level functionality so they can focus on achieving high-level goals
- Dealing with the same concept that must be implemented on many different machines and platforms

```
// Ex.
// C file handles
int open (char * filename,
          int flags);

// Win32 file handles
HFILE OpenFile (
  LPCSTR lpFileName,
  LPOFSTRUCT lpReOpenBuf,
  UINT nStyle);
```

The (Wrapper) Façade Pattern

```
// Ex.
// C file handles
int open (char * filename,
          int flags);

// Win32 file handles
HFILE OpenFile (
  LPCSTR lpFileName,
  LPOFSTRUCT lpReOpenBuf,
  UINT nStyle);
```

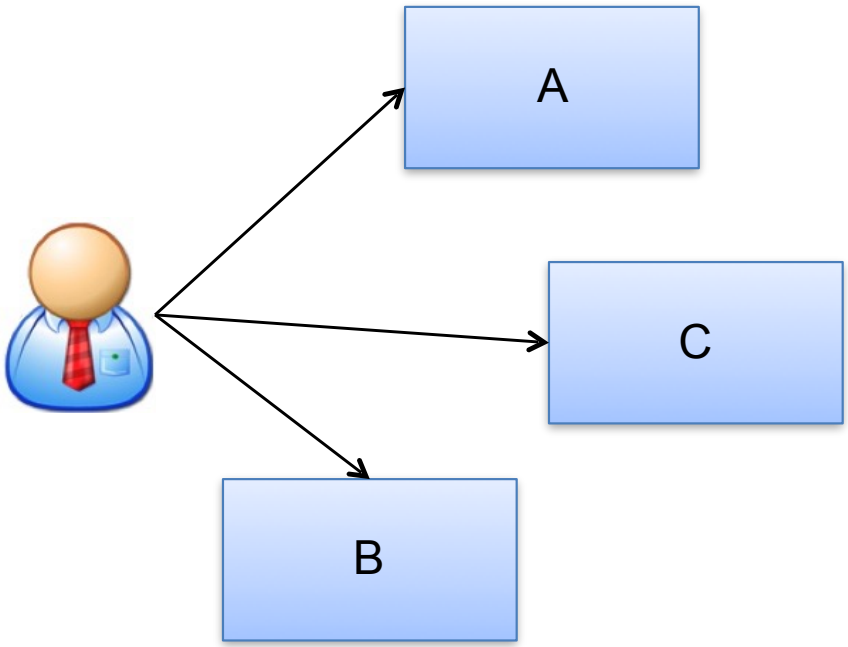
Ideally, we would like to have a single *interface that is consistent on different platforms*

```
// Ex.
// file handles
int open (char * filename, int flags);

// Implement this method in terms of
// open (...) and OpenFile (...)
```

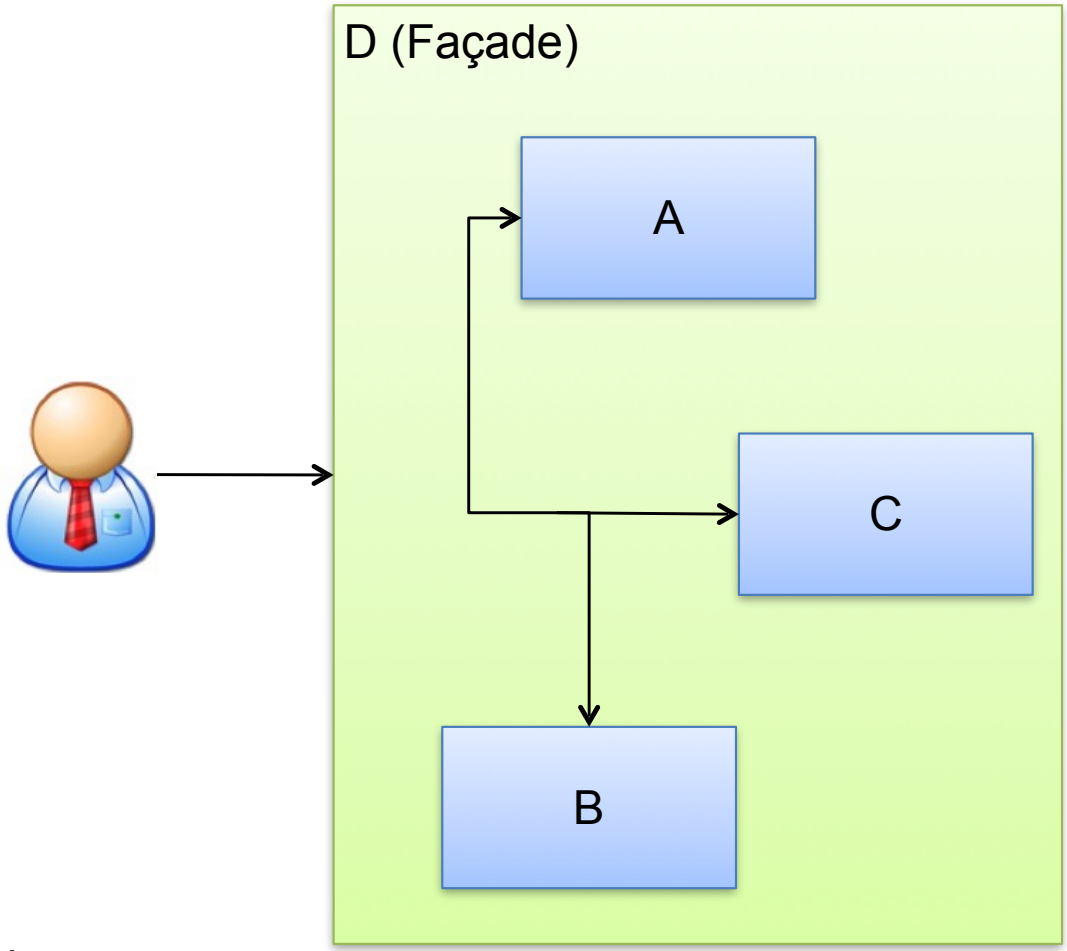
The (Wrapper) Façade Pattern

Intent

Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easy to use



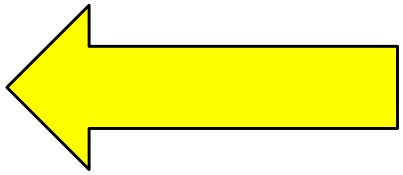
Direct interaction can be hard to correctly unless you possess the necessary domain knowledge it



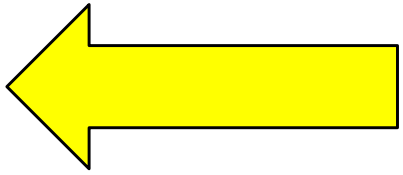
The Façade makes it easier to interact with the subsystems

Example Code Not Using Façade Pattern

```
int main(int argc, char * argv []) {
  char a1[3] = {'C', 'S', 'I'};

  // find the 'C'
  int i = 0;
  for (i = 1; i <= 3; ++ i)
    if (a1[i] == 'C')
      break;

  char * a2 = new char[5];

 // find the 'I' in a1
  int i = 0;
  for (i = 0; i <= 5; ++ i)
    if (a1[i] == 'C')
      break;
}
```



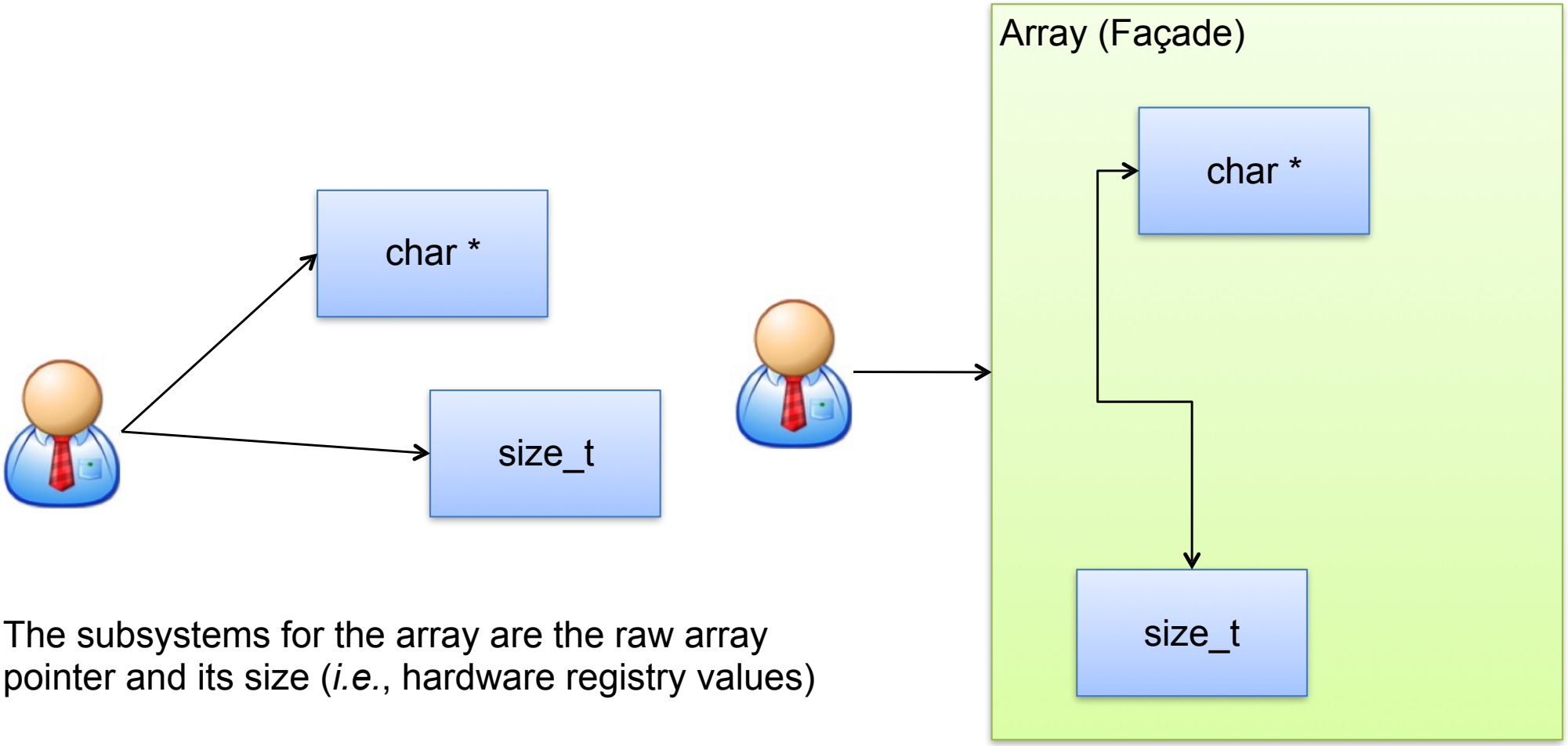
Developer is responsible for knowing that array's use 0-based indexing



Developer is responsible *manually* keeping track of what size belongs to what array (*e.g.*, a1 and a2)

Using the Façade Pattern for C++ Arrays

A better solution would be to provide a Wrapper Façade for working with raw C++ arrays



The subsystems for the array are the raw array pointer and its size (*i.e.*, hardware registry values)

The Façade makes it easier to interact with the subsystems

Using the Façade Pattern for C++ Arrays

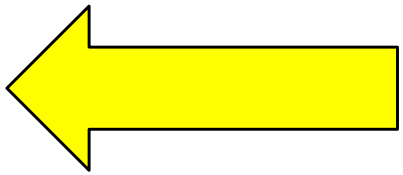
A better solution would be to provide a Wrapper Façade for working with raw C++ arrays

```
class Array {
public:
  Array (void);
  Array (const Array & src);
  ~Array (void);

  void resize (void);
  int find (char ch);

  char & operator [] (size_t i);
  const char & operator [] (size_t i) const;

  bool operator == (const Array & rhs) const;

  // ...
private:
  // the subsystems (just happen to be built-in types)
  char * data_;
  size_t cur_size_;
};
```

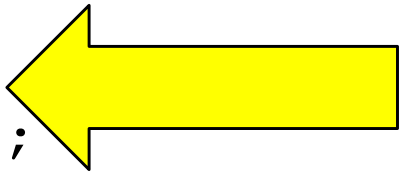
The member functions of this façade know and understand how to interact with the contained subsystems

The (Wrapper) Façade Pattern Example

```
int main(int argc, char * argv []) {
  // create an array of size 3
  // initialize contents to "CSI"
  Array a1 ("CSI", 3);

  // find the 'C'
  size_t pos = a1.find ('C');

  Array a2 (5);

  // find the 'I' in a1
  size_t pos = a1.find ('I');
}
```



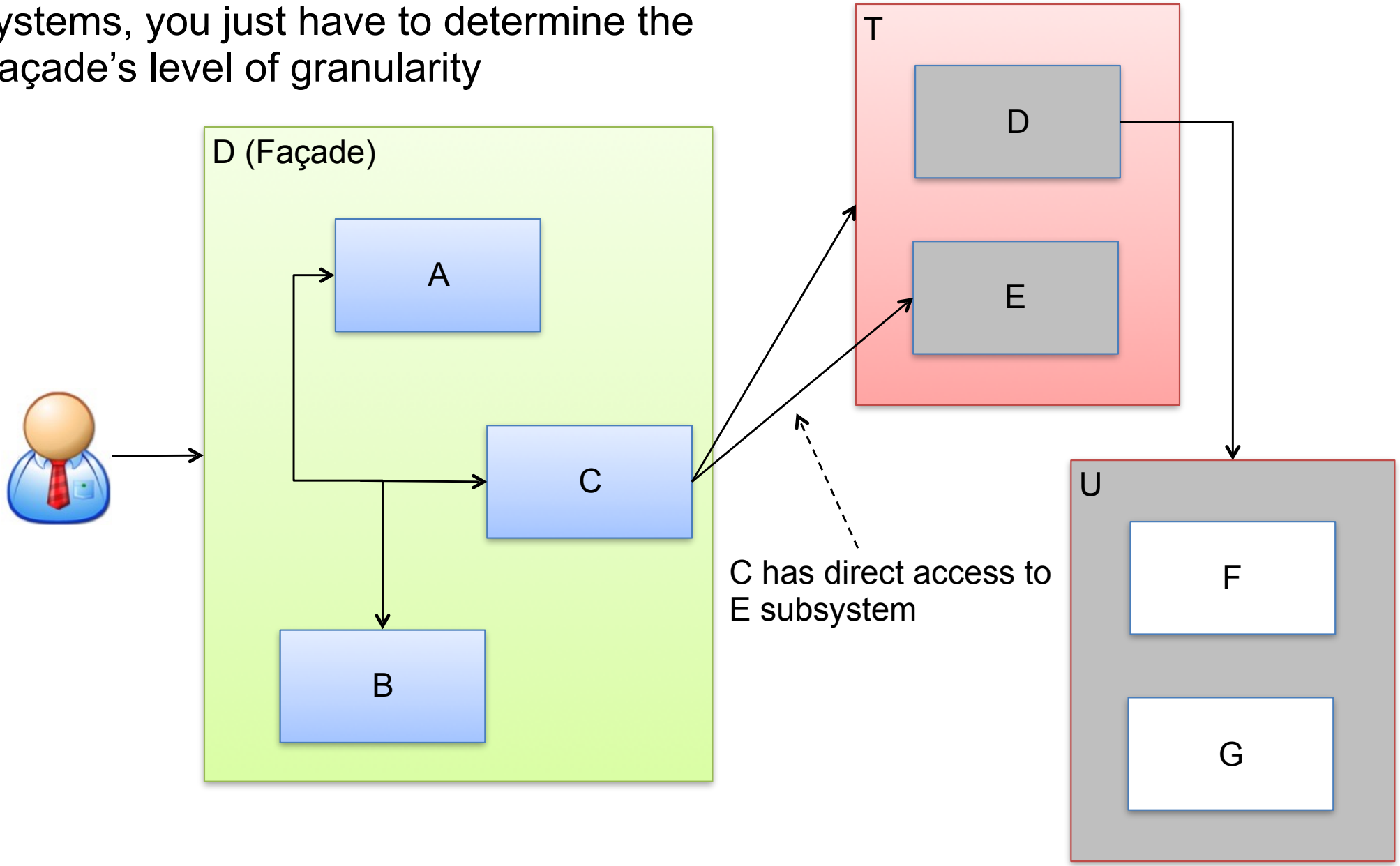
Developer writes less code, & does not have to be concerned with implementing search



No need to manually validate that the correct size is associated with each array

System Composition using the Façade

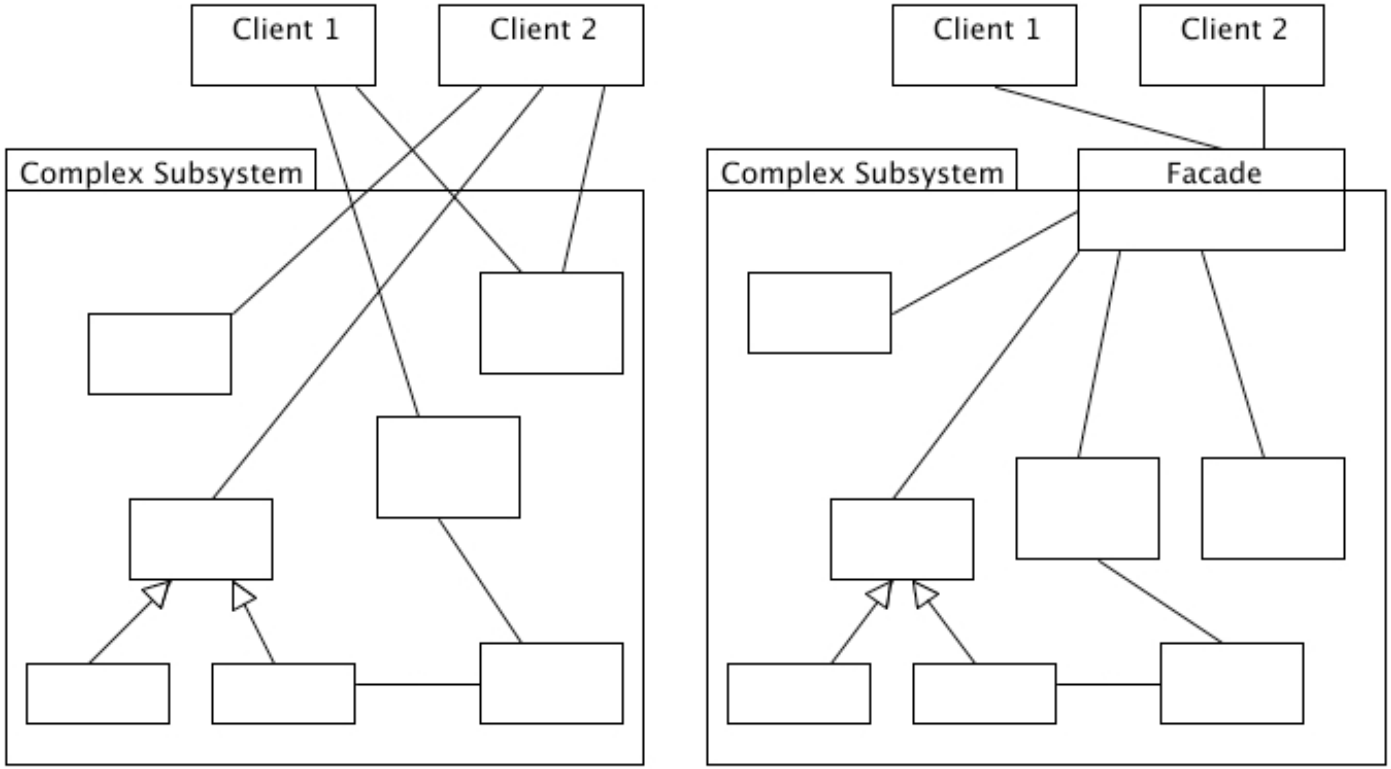
The Façade makes it easier to compose systems, you just have to determine the Façade's level of granularity



The Façade Pattern - Consequences

The following are benefits of the Façade pattern:

- It shields the developers from subsystem components
- Promotes weak coupling between the clients and the subsystem
- Does not prevent applications from using subsystem classes



The Proxy Pattern – Motivation

Have you ever been in the following situation:

- Need to use an object, but really do not care about its location
  - *e.g.*, load a image from disk, memory, network, and etc.
- Have a pointer, but want to add more intelligence to it so it is a LOT easier to use
  - e.g., remembering to delete it once it goes out of scope

The Proxy Pattern – Motivating Example

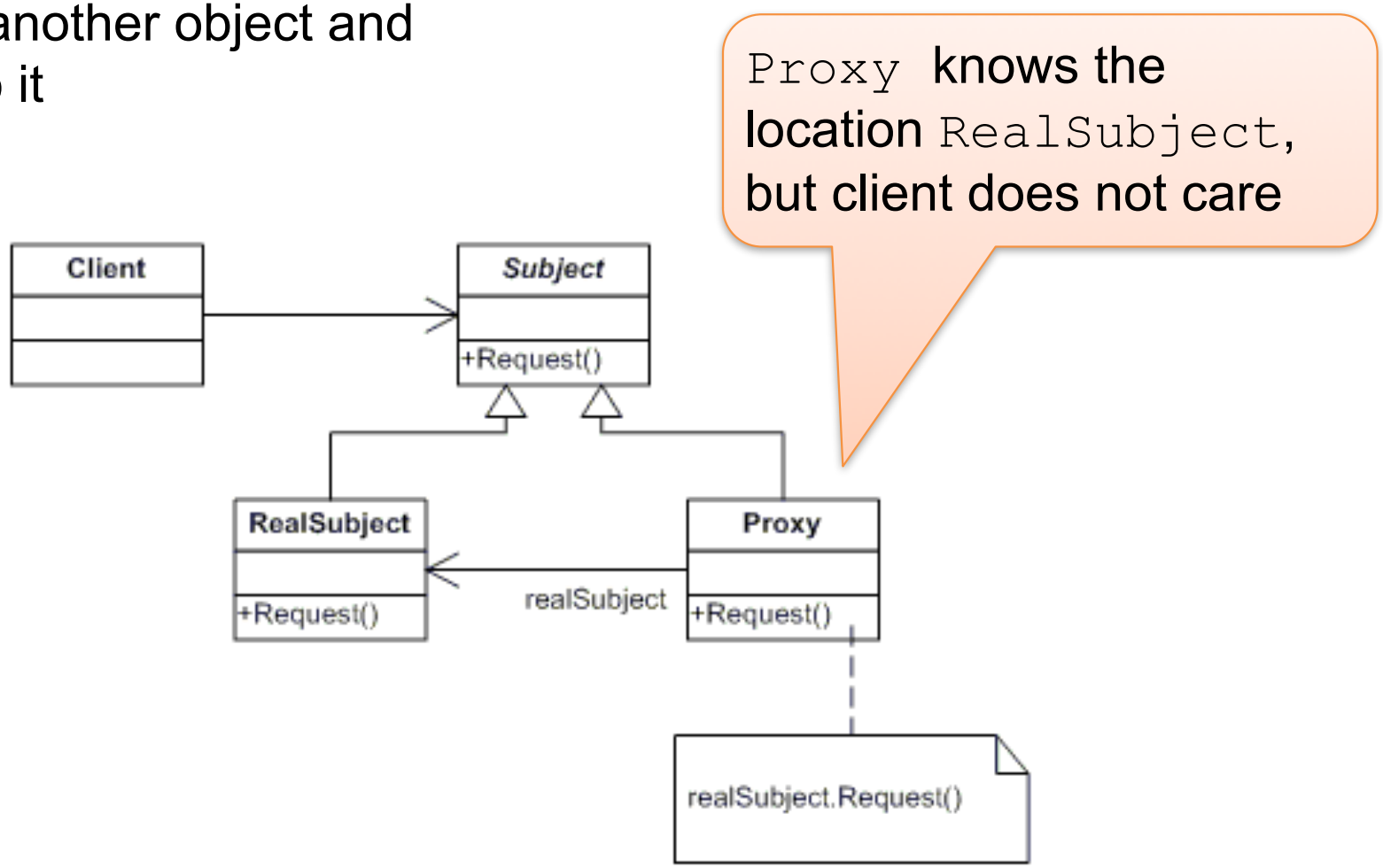
```
int main (int argc, char * argv [])
{
  char ch = 'a';
  char * ch_ptr = &ch;
  const char * my_ch = new char ();
};
```

- Using basic pointers can be a headache. Ideally, we want to have some concept/construct that will know when to delete memory held by a pointer

```
int main (int argc, char * argv [])
{
  char ch = 'a';
  char * ch_ptr = &ch;

  // automatically delete
  pointer my_ch = new char ();
};
```

The Proxy Pattern

Intent
Provide a surrogate, or placeholder, for another object and control access to it



The proxy is meant to look and feel like the real object that it is imitating…

Proxy Pattern Example – Auto Pointers

```
int main (int argc, char * argv [])
{
  char ch = 'a';
  char * ch_ptr = &ch;
  const char * my_ch = new char ();
};
```

- The code above requires developers to manually delete memory when it is out of scope. A better solution would be to automatically delete memory when necessary.

```
int main (int argc, char * argv [])
{
  // ...
  // automatically delete
  pointer my_ch = new char ();
};
```

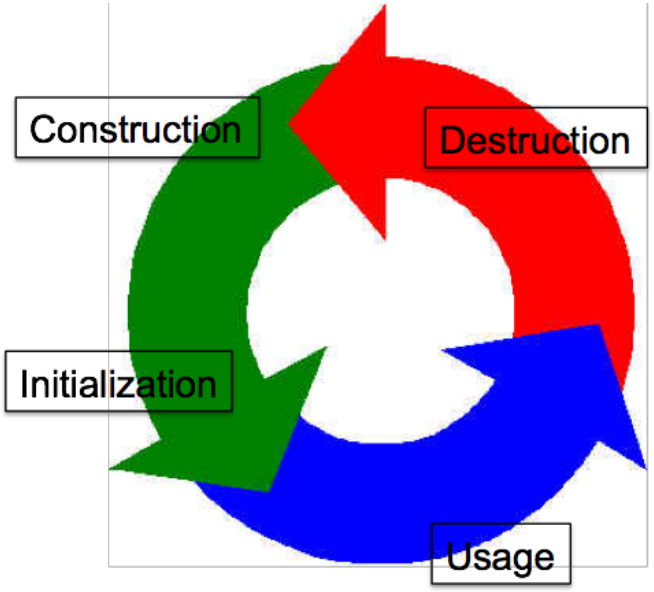
Proxy Pattern Example – Auto Pointers

We can leverage C++ idioms to automatically manage dynamically allocated memory…

```
int main (int argc, char * argv [])
{
  // ...
  // automatically delete
  pointer my_ch = new char ();
};
```

Using the example above, we know the following about `pointer`:

- Its *constructor* will be called only once when the object is constructed
- Its *destructor* will be called only once when the object is destroyed



Object Lifecycle

Proxy Pattern Example – Auto Pointers

We can leverage C++ idioms to automatically manage dynamically allocated memory…

```
template <typename T>
class auto_ptr
{
public:
  // constructors
  auto_ptr (void);
  auto_ptr (T * ptr);

  // desturctor
  ~auto_ptr (void);

  // operator overloading
  T * operator -> (void);
  T & operator * (void);
  operator T * (void);

private:
  // pointer to the memory
  T * ptr_;
};
```

Can use constructor & destructor semantics to manage memory

Proxy Pattern Example – Auto Pointers

We can leverage C++ idioms to automatically manage dynamically allocated memory…

```
template <typename T>
class auto_ptr
{
public:
  // constructors
  auto_ptr (void);
  auto_ptr (T * ptr);

  // desturctor
  ~auto_ptr (void);

  // operator overloading
  T * operator -> (void);
  T & operator * (void);
  operator T * (void);

private:
  // pointer to the memory
  T * ptr_;
};
```

Overloading these operators allows `auto_ptr` to look and feel like a pointer

Implementing Auto Pointers in C++

```
template <typename T>
class auto_ptr
{
public:
  // constructors
  auto_ptr (void)
    : ptr_ (0) { }

  auto_ptr (T * ptr)
    : ptr_ (ptr) { }

  // desturctor
  ~auto_ptr (void) {
     if (0 != this->ptr_)
       delete this->ptr_;
  }

  T * operator -> (void) { return this->ptr_; }
  T & operator * (void) { return *this->ptr_; }
  operator T * (void) { return this->ptr_; }

private:
  // pointer to the memory
  T * ptr_;
};
```

The constructor initializes the contained `ptr_` accordingly.

The destructor automatically deletes memory owned by `ptr_`

The overloaded operators enable access to the encapsulated `ptr_` and its memory

Using Auto Pointers in C++

```
void do_something (void)
{
    int * int_ptr = new int ();

    // ...
    *int_ptr = 5

    // delete memory when done
    delete int_ptr;
}
```

Question

What is one of the major problems with the code above when managing memory?

Using Smart Pointers for Exceptions

- By using an `auto_ptr` to manage memory, if control returns abruptly from `do_something`, then we are guaranteed that the memory will be deleted
  - *i.e.*, no memory leak

```
void do_something (void)
{
   auto_ptr <int> int_ptr = new int ();

   // using overloaded operator *
   *int_ptr = 5

   // int_ptr destructor automatically
   // deletes memory
}
```

Using Smart Pointers in Array Class

The `auto_ptr` also can be use with abstract data types (ADTs):

```
// ADT of type Point
class Point {
public:
  Point (void);
  ~Point (void);

  // ...
  void shift (int dx, int dy);

private:
  int x_, y_;
};


void do_something (void)
{
   auto_ptr <Point> point = new Point ();

   // using overloaded operator ->
   point->shift (5, -7);

   // int_ptr destructor automatically
   // delete memory
}
```

Using Smart Pointers in Array Class

```
class Array
{
  // ...
private:
  char * data_;
  size_t cur_size_;
  size_t max_size_;
};
```

Question

Given the implementation of `auto_ptr`, is it suitable "as is" for usage in the Array class, and why is this the case?

The Array Auto Pointer

The `array_auto_ptr` has the same intention as `auto_ptr`, but its semantics are designed for C++ arrays

```
template <typename T>
class array_auto_ptr
{
public:
  // constructors
  array_auto_ptr (void);
  array_auto_ptr (T * ptr);

  // desturctor
  ~array_auto_ptr (void);

  // operator overloading
  T * operator -> (void);
  T & operator * (void);
  operator T * (void);
  T & operator [] (int i);

private:
  // pointer to the memory
  T * ptr_;
};
```

Same look and feel as auto_ptr, but overloads the bracket operator

The Array Auto Pointer

```
template <typename T>
class array_auto_ptr {
public:
  // constructors
  array_auto_ptr (void)
    : ptr_ (0) { }

  array_auto_ptr (T * ptr)
    : ptr_ (ptr) { }

  // desturctor
  ~array_auto_ptr (void) {
     if (0 != this->ptr_)
       delete [] this->ptr_;
  }

  T * operator -> (void) { return this->ptr_; }
  T & operator * (void) { return *this->ptr_; }
  operator T * (void) { return this->ptr_; }

  // index into array
  T & operator [] (int i) { return this->ptr_[i]; }

private:
  // pointer to the memory
  T * ptr_;
};
```

The destructor invokes the correct version of the delete operator

Improving the Array Class

We can now us the `array_auto_ptr` to manage memory allocations of the `Array` class to improve its quality

```
class Array
{
  ~Array (void) {
    // array_auto_ptr handles delete for you!!
  }

private:
  array_auto_ptr <char> data_;
  size_t cur_size_;
  size_t max_size_;
};
```

Smart Pointers in C++11

The C++11 specification introduces 3 new smart pointers for managing memory allocations:

- **unique_ptr** – a smart pointer that retains sole ownership of an object through a pointer

- **shared_ptr** – a smart pointer that manages lifetime of an object, typically allocated with new

- **weak_ptr** – a smart pointer that holds a non-owning ("weak") reference to an object that is managed by std::shared_ptr. It must be converted to std::shared_ptr in order to access the referenced object

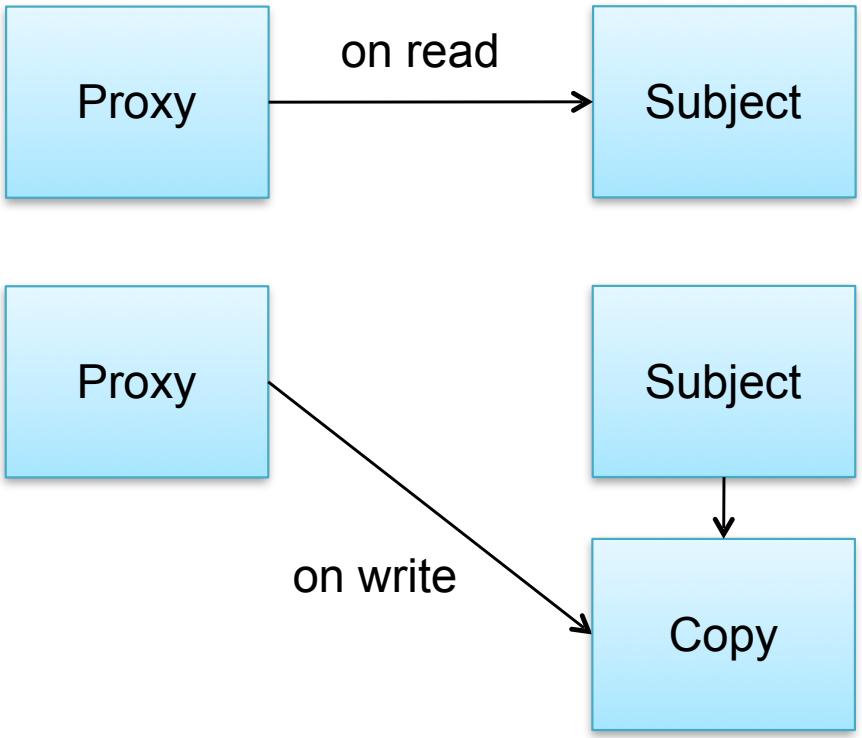
Deprecation Notice: auto_ptr has been deprecated from the C++ standard. It should be replaced by one of the smart pointers above.

The Proxy Pattern - Consequences

The following are some consequences of the Proxy pattern:

- Introduces a level of indirection when accessing the underlying object

- Copy-on-write, which is the idea of copying an object only after you begin to write to it
  - E.g., can reduce the cost of copying heavyweight objects



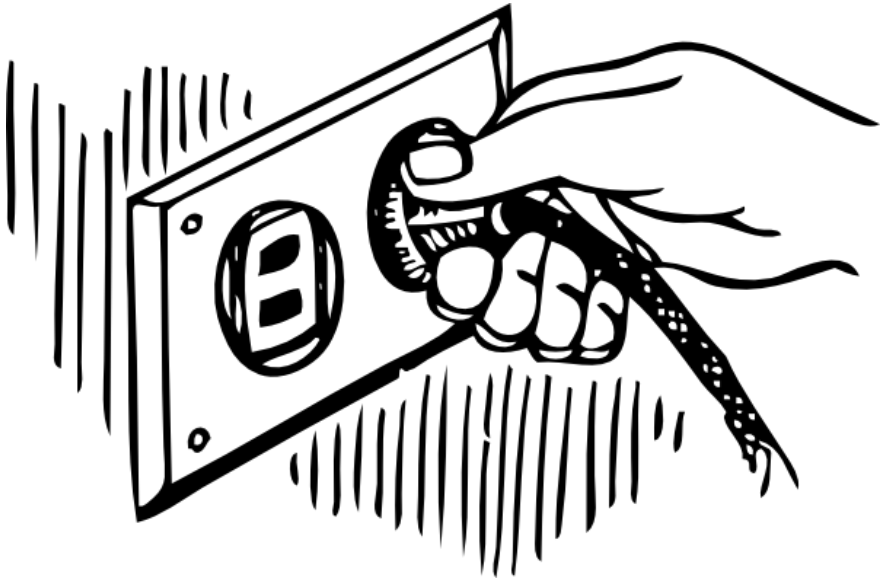
After first operation that modifies state is invoked, a copy is created and remaining invocations happen on copy

The Adapter Pattern

Have you ever been in the following situation:

- Have a method that expects an certain interface, and a class that does not have the expected interface
    - *e.g.* a class that is not designed to work with a template method

Problem Analogy
Have you ever been to a foreign country, tried to plug in an electronic device (such as a laptop), & realized the interfaces doe not match?



Expected Interface(s)

The Adapter Pattern – Motivating Example

Assume you found a “third-party” implementation of an event handler

```
template <typename T>
class Event_Handler {
public:
  Event_Handler (Queue <T> & queue)
    : queue_ (queue) { }

  ~Event_Handler (void);

  int open (void);
  int close (void);

  void handle_event (T item);
  void reset (void);

private:
  int event_handler_thread (void);

  /// The event queue for the event handler
  Queue <T> event_queue_;
};
```

Requires an Queue for processing events

The Adapter Pattern – Motivating Example

Assume you found a “third-party” implementation of an event handler

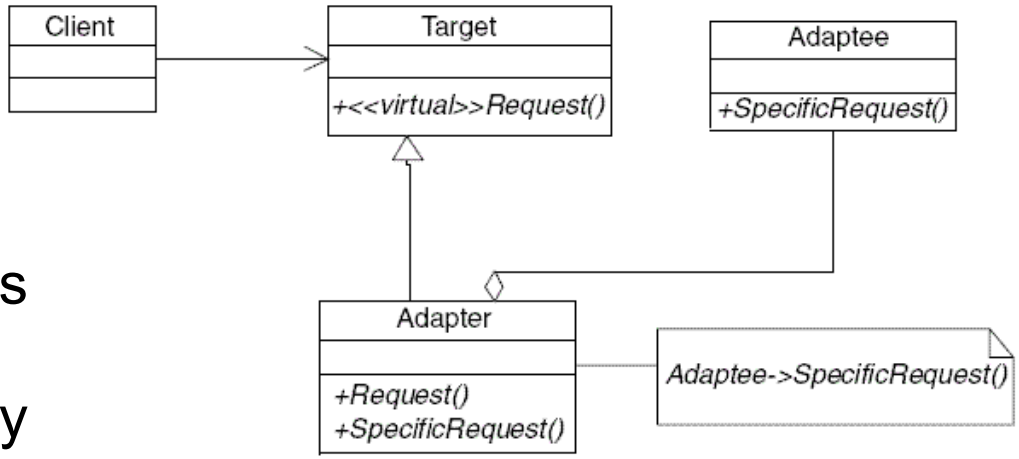
```
template <typename T>
class Event_Handler {
public:
  Event_Handler (Queue <T> & queue)
    : queue_ (queue) { }

  ~Event_Handler (void);

  int open (void);
  int close (void);

  void handle_event (T item);
  void reset (void);

private:
  int event_handler_thread (void);

  /// The event queue for the event handler
  Queue <T> event_queue_;
};
```

Thus far, we can have an Array, but the function expects a Queue!

The Adapter Pattern

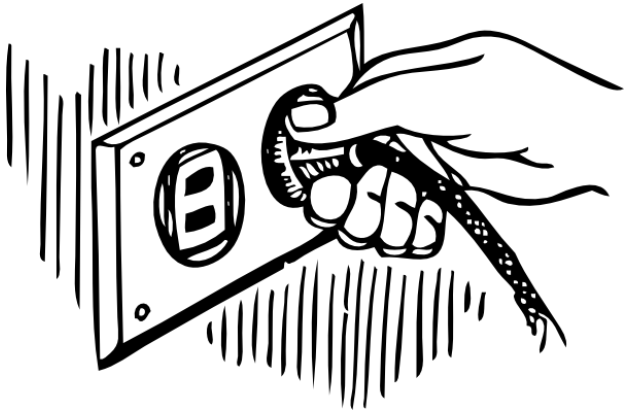
Intent

Convert an interface of a class to another interface the client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



Pattern Analogy

The best way to think of an adapter is to think of how you have to use an adapter for a plug in a foreign country



Expected Interface

Adapters for Creating Expected Interface

The Adapter Pattern Example – The Knowns

- Right now, we know what the Event_Handler class needs a Queue to function correctly

```
template <typename T> class Event_Handler
```

- We have implemented a simple array class for handling many different data types

```
typedef Array <int> Int_Array;
```

The challenge is adapting the Array class to work with Event_Handler function

The Adapter Pattern Example

From investigating the method, we learn the expected interface for each of the methods:

```
template <typename T>
int Event_Handler <T>::close (void) { this->reset (); }

template <typename T>
void Event_Handler <T>::handle_event (T item) {
  this->event_queue_.enqueue (item);
}

template <typename T>
void Event_Handler <T>::reset (void) {
  this->event_queue_.clear ();
}

template <typename T>
int Event_Handler<T>::event_handler_thread (void) {
  while (this->is_open_) {
    T item = this->event_queue_.dequeue ();

    // process the event
  }
}
```

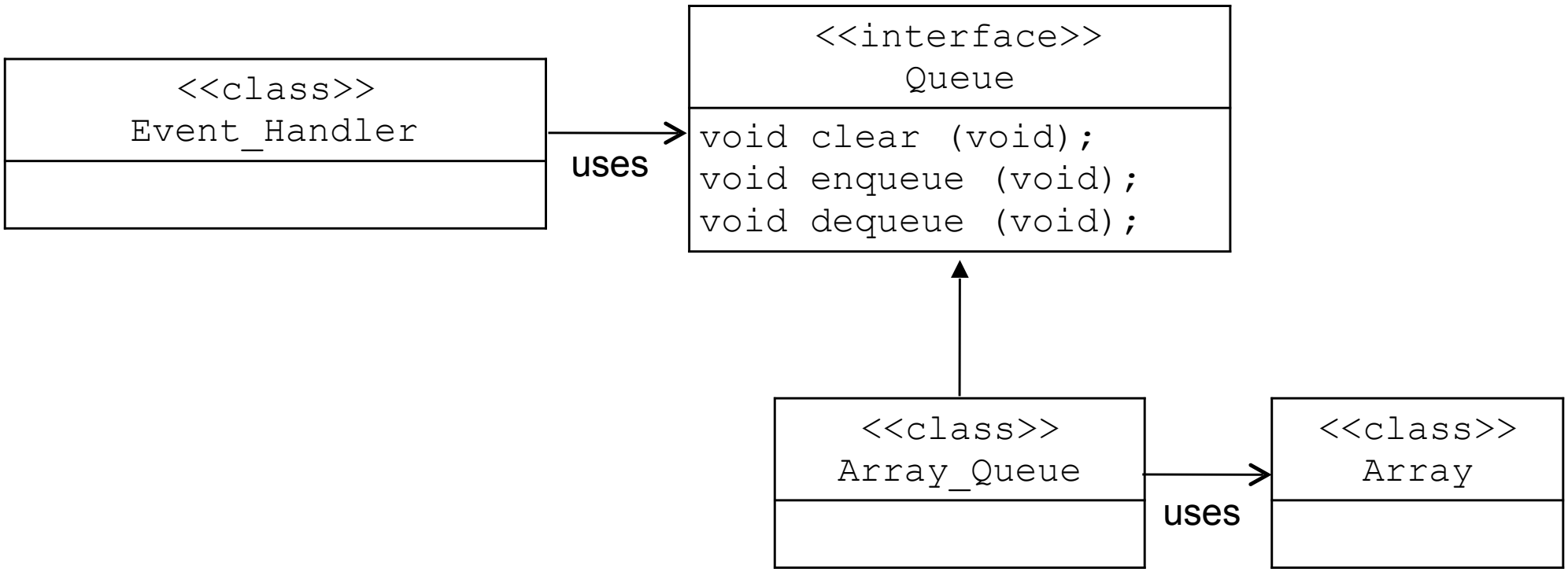
The Adapter Pattern Example

The expected interface for the `Queue` used in `Event_Handler` is as follows:

```
template <typename T>
class Queue
{
  void clear (void);
  void enqueue (T & item);
  T dequeue (void);
};
```

Typically, it is the responsibility of the framework to provide this interface. It is the responsibility of the client to use the interface by adapting their code implement the interface

The Adapter Pattern Example



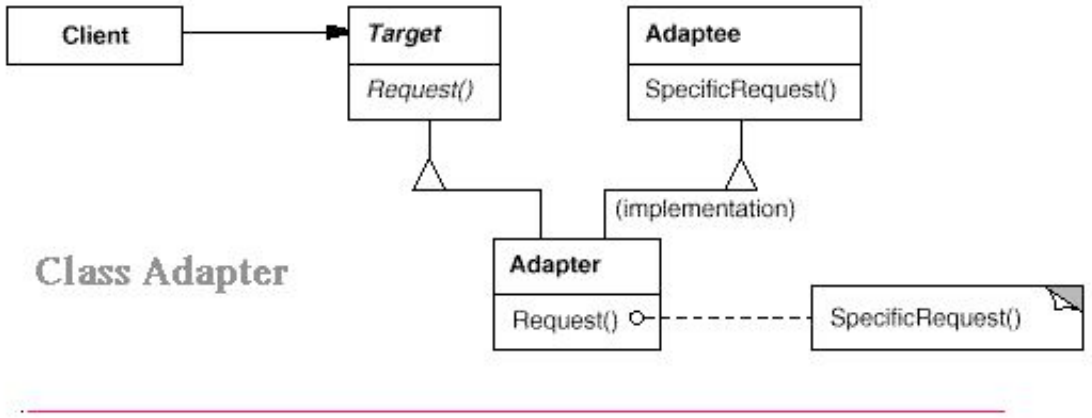
The Array_Queue is designed *to adapt* the Array class to work with the client (*i.e.*, Event_Handler)

The Adapter Pattern – Consequences

The Adapter class has the following consequences:

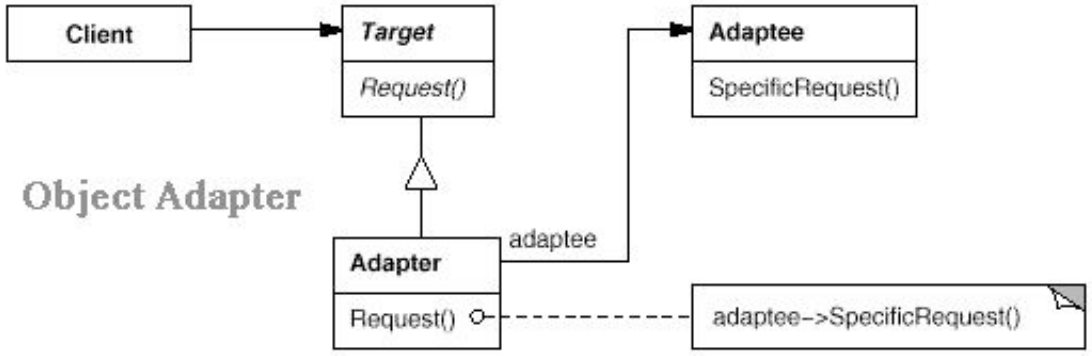
Class Adapter Consequences

- Adapts Adaptee to Target by committing Concrete class
- Lets Adapter override some of Adaptee's behavior
- No additional pointer indirection is needed



Object Adapter Consequences

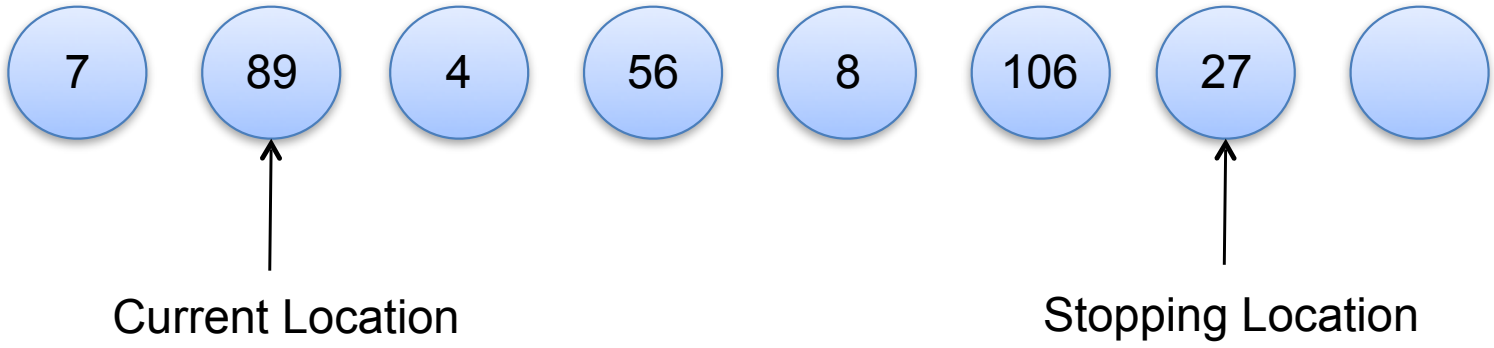
- Let's a single Adapter work with many Adaptees
  - i.e., the Adaptee itself and all its subclasses
- Makes it hard to override Adaptee behavior



The Iterator Pattern

Have you ever been in the following situation:

- Have a collection of elements, and need to access them in a sequential manner
  - *e.g.* an array of Point objects or a set of integers



The Iterator Pattern – Motivating Example

Using the index operator in client code:

```
Array <int> a(56);

// ...
for (size_t i = 0; i < a.size (); ++ i)
  cout << '[' << a[i] << ']';

// ...
```

Uses operator [] to access each element

The Iterator Pattern – Motivating Example

Using the index operator in client code:

```
Array <int> a(56);

// ...
for (size_t i = 0; i < a.size (); ++ i)
  cout << '[' << a[i] << ']';

// ...
```

Uses operator [] to access each element

Code for overloading the index operator for Array:

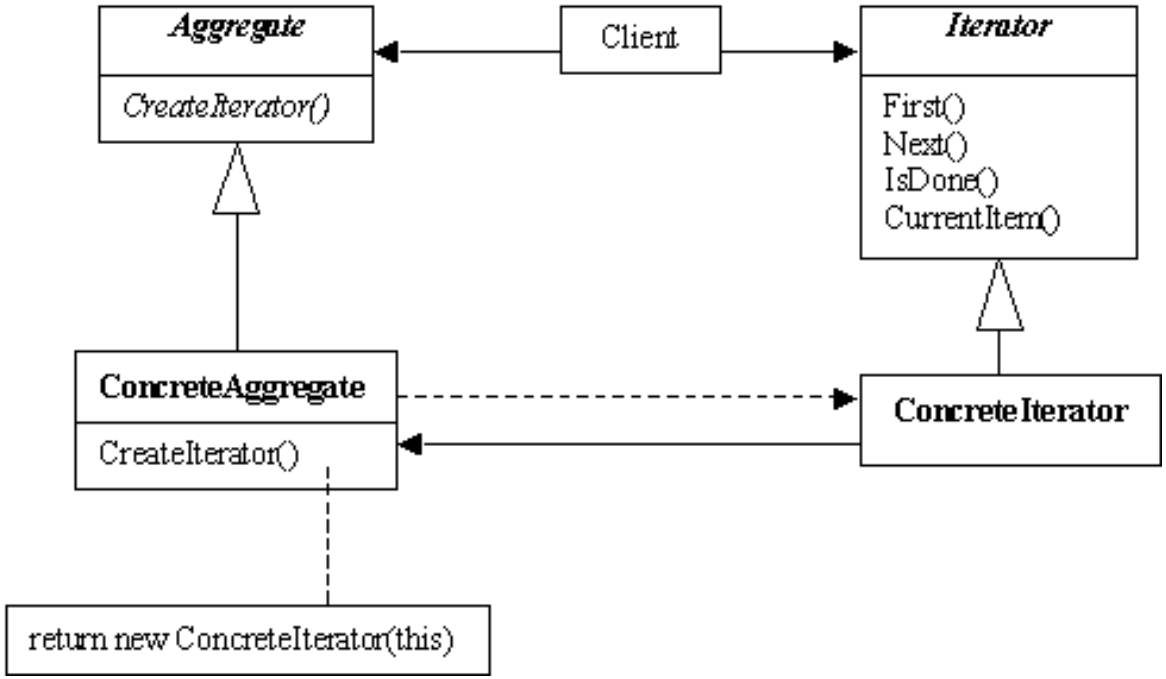
```
template <typename T>
T & Array <T>::operator [] (size_t i)
{
  if (i < this->cur_size_)
    return this->data_[i];

  throw std::out_of_range ("in
}
```

Accessing each element results in unnecessary bound check

The Iterator Pattern

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing the underlying representation



The Iterator Pattern Example

Up to this point, we have defined an Array class that contains a set of elements:

```
template <typename T>
class Array {
  Array (void);
  // ...

private:
  T * data_;
  size_t cur_size_;
  size_t max_size_;
};
```

We would like to define an ADT that can iterate over the elements without penalizing the client for accessing the elements.

The Iterator Pattern Example

First, we defined an ADT called Array_Iterator whose purpose is to access each element in the Array class

```
template <typename T>
class Array_Iterator {
public:
  Array_Iterator (Array <T> & a)
    : a_ (a), curr_ (0) { }
  ~Array_Iterator (void)

  bool is_done (void) { return this->curr_ >= this->a_.cur_size_; }
  bool advance (void) { ++ this->curr_; }
  T & operator * (void) { return this->a.data_[this->curr_]; }
  T * operator -> (void) { return &this->a_.data_[this->curr_]; }

private:
  Array <T> & a_;
  size_t curr_;

};
```

The Array_Iterator can access to Array class's private variables because it is a friend class of the Array class. In Java and C#, this is possible if Array_Iterator is in the same package as the Array class.

The Iterator Pattern Example

Using the Array_Iterator class in client code:

```
typedef Array <int> Int_Array;
Int_Array a1 (7, 'z');

a1[0] = 'J';
a1[1] = 'o';
a1[2] = 'h';
a1[3] = 'n';

typedef Array_Iterator <int> Int_Array_Iterator;

for (Int_Array_Iterator iter (a1); !iter.is_done (); iter.advance ())
  std::cout << '[' << (char)*iter << ']';
```

Declare an array of an initial size and fill value

The Iterator Pattern Example

Using the Array_Iterator class in client code:

```
typedef Array <int> Int_Array;
Int_Array a1 (7, 'z');

a1[0] = 'J';
a1[1] = 'o';
a1[2] = 'h';
a1[3] = 'n';

typedef Array_Iterator <int> Int_Array_Iterator;

for (Int_Array_Iterator iter (a1); !iter.is_done (); iter.advance ())
  std::cout << '[' << (char)*iter << ']';
```

Set the value of several elements in the array

The Iterator Pattern Example

Using the Array_Iterator class in client code:

```
typedef Array <int> Int_Array;
Int_Array a1 (7, 'z');

a1[0] = 'J';
a1[1] = 'o';
a1[2] = 'h';
a1[3] = 'n';

typedef Array_Iterator <int> Int_Array_Iterator;

for (Int_Array_Iterator iter (a1); !iter.is_done (); iter.advance ())
  std::cout << '[' << (char)*iter << ']';
```

Declare an iterator for the array

The Iterator Pattern Example

Using the Array_Iterator class in client code:

```
typedef Array <int> Int_Array;
Int_Array a1 (7, 'z');

a1[0] = 'J';
a1[1] = 'o';
a1[2] = 'h';
a1[3] = 'n';

typedef Array_Iterator <int> Int_Array_Iterator;

for (Int_Array_Iterator iter (a1); !iter.is_done (); iter.advance ())
  std::cout << '[' << (char)*iter << ']';
```

Loop over the contents of the array

The Iterator Pattern Example

Using the Array_Iterator class in client code:

```
typedef Array <int> Int_Array;
Int_Array a1 (7, 'z');

a1[0] = 'J';
a1[1] = 'o';
a1[2] = 'h';
a1[3] = 'n';

typedef Array_Iterator <int>  _Array_Iterator;

for (Int_Array_Iterator ite  (a1); !iter.is_done (); iter.advance ())
  std::cout << '[' << (char)*iter << ']';
```

Print the value of each element

The Iterator Pattern Example

Using the Array_Iterator class in client code:

```
typedef Array <int> Int_Array;
Int_Array a1 (7, 'z');

a1[0] = 'J';
a1[1] = 'o';
a1[2] = 'h';
a1[3] = 'n';

typedef Array_Iterator <int> Int_Array_Iterator;

for (Int_Array_Iterator iter (a1); !iter.is_done (); iter.advance ())
  std::cout << '[' << (char)*iter << ']';
```

Print the value of each element

The code above prints the following code:

```
[J][o][h][n][z][z][z]
```

The Iterator Pattern Example

The code has the following advantages:

```
typedef Array <int> Int_Array;
Int_Array a1 (7, 'z');

a1[0] = 'J';
a1[1] = 'o';
a1[2] = 'h';
a1[3] = 'n';

typedef Array_Iterator <int>    _Array_Iterator;

for (Int_Array_Iterator iter (a1); !iter.is_done (); iter.advance ())
  std::cout << '[' << (char)*iter << ']';
```

Iterator hides the array's implementation

The Iterator Pattern Example

The code has the following advantages:

```
typedef Array <int> Int_Array;
Int_Array a1 (7, 'z');

a1[0] = 'J';
a1[1] = 'o';
a1[2] = 'h';
a1[3] = 'n';

typedef Array_Iterator <int> I     ray_Iterator;

for (Int_Array_Iterator ite     (a1); !iter.is_done (); iter.advance ())
  std::cout << '[' << (cha  )*iter << ']';
```

Does not result in unnecessary bounds check when accessing elements

The Iterator Pattern - Consequences

The Iterator Patterns has the following consequences:

1. It supports variations in the traversal of an aggregate
   - e.g., in-order vs. reverse order of a collection
2. Iterators simplify the Aggregate interface
3. More than one traversal can be pending on a aggregate
   - i.e., each Iterator keeps track of its own state

```
template <typename IterT>
std::for_each (IterT begin, IterT end);
```

The STL std::for_each function does not care about actual iterator type

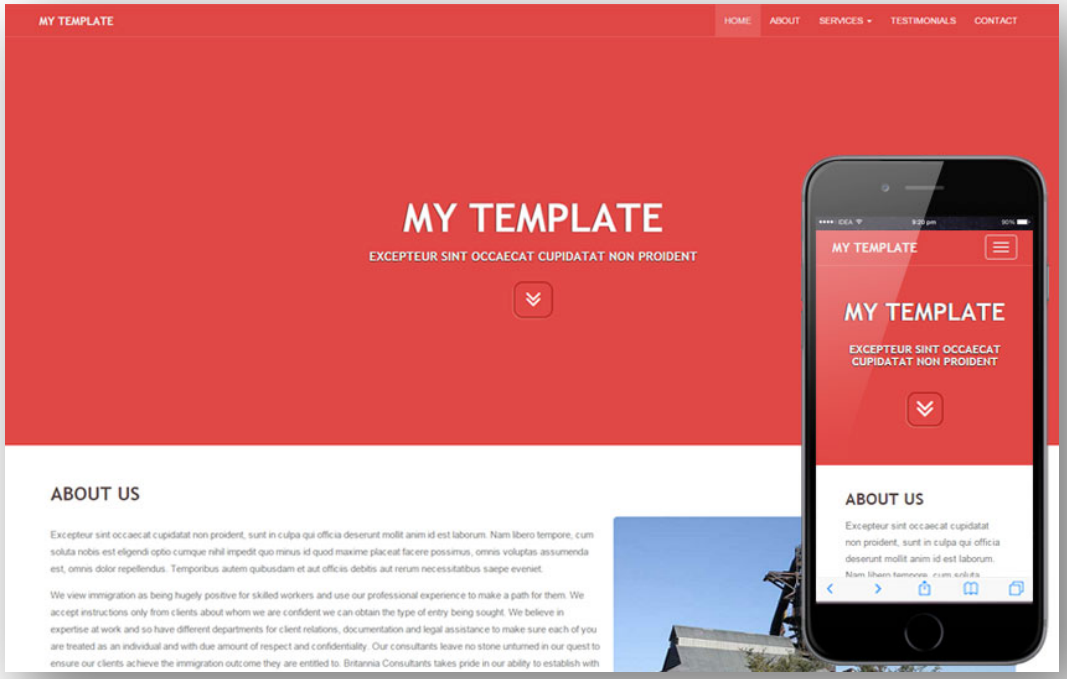
```
Int_Array_Iterator i1 (a1);
Int_Array_Iterator i2 (a1);
// ...

for (; !i1.is_done (); i1.advance ())
  cout << '[' << *i1 << ']';
```

The state of i2 is not affected by i1.

The Template Method Pattern - Motivation

Have you ever been in the following situation:

- Have an algorithm that you would like to generalize and apply to many different data types or situations
    - *e.g.* finding an element, viewing a document, parsing input
- Need to provide common (or well-defined) behavior to a set of subclasses
    - e.g., creating an new document, initialize itself



The Template Method Pattern – Motivate Example

```
int find (char * buffer, size_t n, char val) {
  char * end = buffer + n;
  for (char * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
};
```

Using conventional approaches, if we want to use the `find ()` function on other data types, then we must create a duplicate copy of the method.

```
int find (int * buffer, size_t n, int val) {
  int * end = buffer + n;
  for (int * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
};
```

The Template Method Pattern – Motivate Example

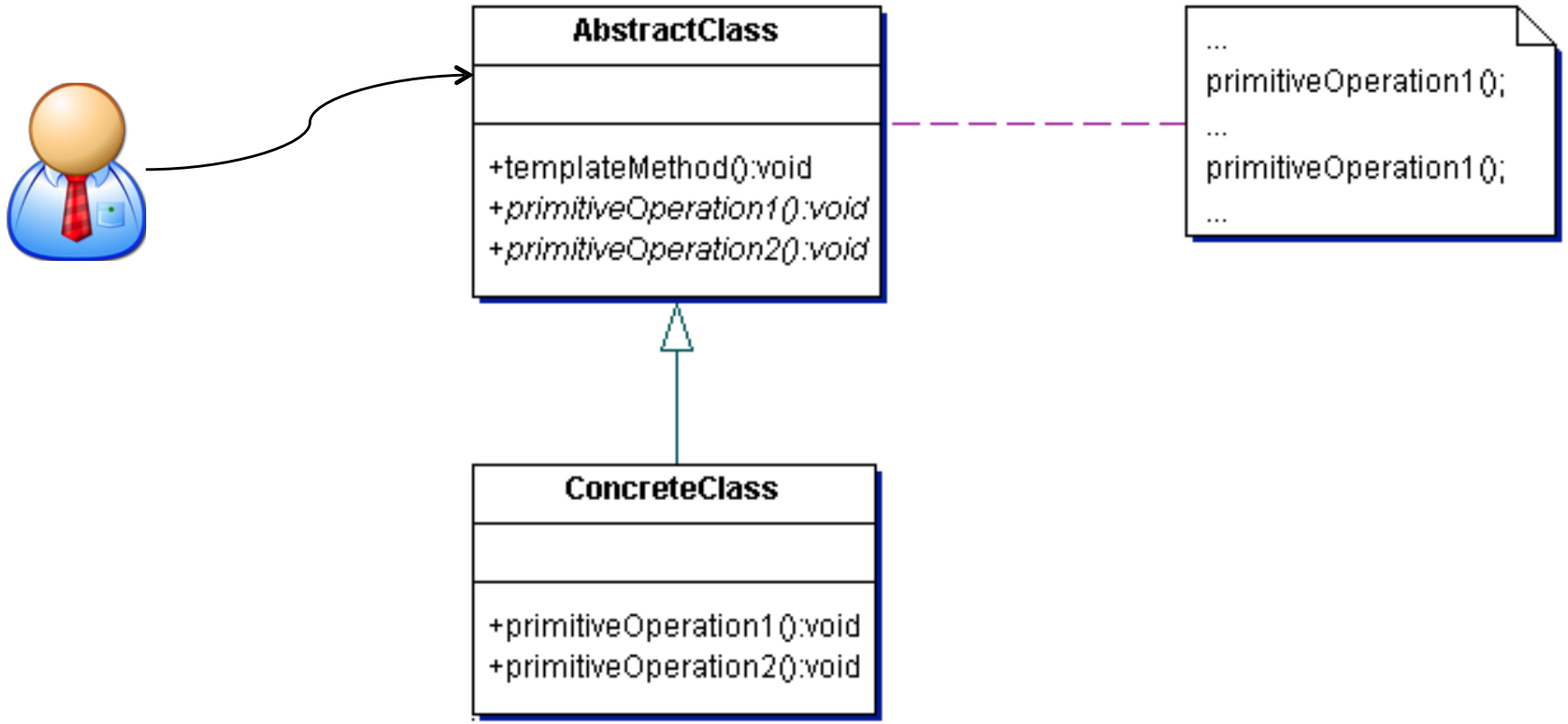
```
int find (char * buffer, size_t n, char val) {
  char * end = buffer + n;
  for (char * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
};
```

By using conventional approaches to “replicate” an algorithm, what principles of rotten design are we adding to the code base?

The Template Method Pattern

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms structure.



The Template Method using Inheritance

```
bool find (Iterator & iter, char val) {
  while (!iter.is_done ()) {
    ch = iter.get_next ();

    if (ch == val)
      return true;
    iter.advance ();
  }

  return false;
};
```

The Template Method expects the type to implement `is_done ()`

- Assume that `Iterator` is an ADT that can knows how to access the elements of a collection, such as an `Array`.

The Template Method using Inheritance

```
bool find (Iterator & iter, char val) {
  while (!iter.is_done ()) {
    ch = iter.get_next ();

    if (ch == val)
      return true;
    iter.advance ();
  }

  return false;
};
```

The Template Method expects the type to implement `get_next ()`

- Assume that `Iterator` is an ADT that can knows how to access the elements of a collection, such as an `Array`.

The Template Method using Inheritance

```
bool find (Iterator & iter, char val) {
  while (!iter.is_done ()) {
    ch = iter.get_next ();

    if (ch == val)
      return true;
    iter.advance ();
  }

  return false;
};
```

The Template Method expects the type to implement `advance ()`

- Assume that `Iterator` is an ADT that can knows how to access the elements of a collection, such as an `Array`.

The Template Method using Inheritance

```
bool find (Iterator & iter, char val) {
  while (!iter.is_done ()) {
    ch = iter.get_next ();

    if (ch == val)
      return true;
    iter.advance ();
  }

  return false;
};
```

- Given the following algorithm named `find ()`, the expected interface for `Iterator` is as follows:

```
class Iterator {
  // ...

  virtual bool is_done (void) const = 0;
  virtual char get_next (void) const = 0;
  virtual void advance (void) = 0;

  // ...
};
```

This is the contact for being compatible with the `find ()` function

The Template Method using C++ Templates

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
};
```

The template using C++ templates is slightly easier since the *template parameters* hold the place of the overloaded operators.

The Template Method using C++ Templates

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
```

We let `T` hold the place of ever location where the algorithm can vary.

The template using C++ templates is slightly easier since the *template parameters* hold the place of the overloaded operators.

The Template Method using C++ Templates

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
};
```

The template using C++ templates is slightly easier since the *template parameters* hold the place of the overloaded operators.

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if ((*iter).operator == (val))
      return iter - buffer;
  }

  return -1;
};
```

The Template Method using C++ Templates

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
};
```

Although using `T` makes the algorithm look cleaner, under the hood the algorithm expects a certain contract.

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if ((*iter).operator == (val))
      return iter - buffer;
  }

  return -1;
};
```

The compiler checks if the ADT has defined the `operator ==` method

The Template Method using C++ Templates

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
};
```

Although using `T` makes the algorithm look cleaner, under the hood the algorithm expects a certain contract.

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (operator == (*iter, val))
      return iter - buffer;
  }

  return -1;
};
```

The compiler checks for the global comparison operator for the `T`.

The Template Method using C++ Templates

```
template <typename T>
int find (T * buffer, size_t n, T val) {
  T * end = buffer + n;
  for (T * iter = buffer, * end = buffer + n;
       iter < end; ++ iter) {
    if (*iter == val)
      return iter - buffer;
  }

  return -1;
};
```

Given the following algorithm named `find ()`, the expected interface for an ADT (such as `Foo`) for data type `T` is the following

```
class Foo {
  // ...

  bool operator == (const Foo &) const;

  // ...
};
```

`Foo` may have many other methods, but the `find ()` function only cares about `operator ==`

The Template Method using C++ Templates

```
template <typename T>
void Array <T>::resize (size_t new_size) {
// ...
 else if (new_size > this->max_size_)
  {
    // Allocate a new array since we are too small.
    size_t length = new_size;
    T * temp = new T [length];

    // Copy the old elements to the new array.
    for (size_t i = 0; i < this->cur_size_; ++ i)
      temp[i] = this->data_[i];

    // ...

    // Safely delete the old data. ;-)
    delete [] temp;
  }
};
```

Given the `resize ()` method above, what is the expected methods of `T` if `T` is an ADT?

The Template Method – Consequences

- Template methods are a fundamental technique for reuse
- They can lead to inverted control structure
  - i.e., the Hollywood principle, “Don’t call us, we’ll call you”


```
bool find (Iterator & iter, char val) {
  while (!iter.is_done ()) {
    ch = iter.get_next ();

    if (ch == val)
      return true;
    iter.advance ();
  }

  return false;
};
```

The function calls `Iterator` when it is ready…

The Template Method – Consequences

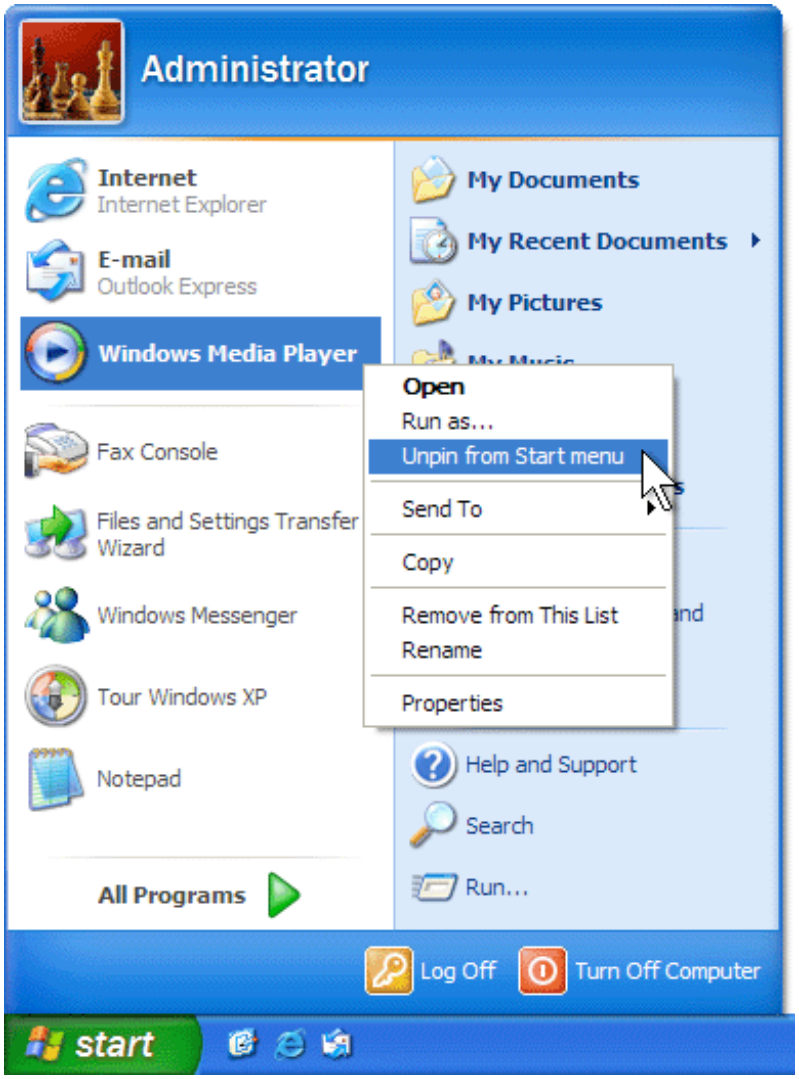
- Template methods are a fundamental technique for reuse
- They can lead to inverted control structure
  - i.e., the Hollywood principle, “Don’t call us, we’ll call you”
- Critical to determine what operations are optional, and what operations must be overridden
  - *e.g.*, virtual vs. abstract, respectively


```
class PersonActions
{
  virtual ~PersonActions (void);
  virtual void go_to_class (void);
  virtual void do_assignments (void);

  // definitely required
  virtual void eat_food (void) = 0;
  // ...
};

void graudate (PersonActions * p);
```

The Command Pattern

Have you ever been in the following situation:

- Need to create an event, but process it at a later time
  - e.g., events in an event queue
- Decouple the handling of many common (or similar operations) from the actual processing
  - i.e., you really do not care about the actual actions
- Need to support undoing of executed operations



The Command Pattern – Motivating Example

Currently, the evaluate_postfix method is defined as follows:

```
bool evaluate_postfix (Expr_Iterator & iter, int & result) {
  Stack <int> s;
  for (; !iter.is_done (); iter.advance ()) {
  switch (iter->tag_)
  case OPERATOR:
    int n1 = r.pop (), n2 = r.pop ();
    switch ((*iter).op_) {
    case '+': r.push (n2 + n1); break;
    case '-': r.push (n2 - n1); break;
    case '*': r.push (n2 * n1); break;
    case '/': r.push (n2 / n1); break;
    }
  case NUM:
    r.push (iter->num_); break;
  default:
    return false;
  }

  result = s.pop ();
  return true;
}
```

The "execution" of each operation is tightly couple with the iteration logic

The Command Pattern – Motivating Example

Currently, the evaluate_postfix method is defined as follows:

```
bool evaluate_postfix (Expr_Iterator & iter, int & result) {
  Stack <int> s;
  for (; !iter.is_done (); iter.advance ()) {
  switch (iter->tag_)
  case OPERATOR:
    int n1 = r.pop (), n2 = r.pop ();
    switch ((*iter).op_) {
    case '+': r.push (n2 + n1); break;
    case '-': r.push (n2 - n1); break;
    case '*': r.push (n2 * n1); break;
    case '/': r.push (n2 / n1); break;
    }
  case NUM:
    r.push (iter->num_); break;
  default:
    return false;
  }

  result = s.pop ();
  return true;
}
```

The evaluator must determine how to handle numbers and operators

The Command Pattern – Motivating Example

Currently, the evaluate_postfix method is defined as follows:

```
bool evaluate_postfix (Expr_Iterator & iter, int & result) {
  Stack <int> s;
  for (; !iter.is_done (); iter.advance ()) {
  switch (iter->tag_)
  case OPERATOR:
    int n1 = r.pop (), n2 = r.pop ();
    switch ((*iter).op_) {
    case '+': r.push (n2 + n1); break;
    case '-': r.push (n2 - n1); break;
    case '*': r.push (n2 * n1); break;
    case '/': r.push (n2 / n1); break;
    }
  case NUM:
    r.push (iter->num_); break;
  default:
    return false;
  }

  result = s.pop ();
  return true;
}
```

The evaluator must determine how to deal with invalid “tags”

The Command Pattern – Motivating Example

Currently, the evaluate_postfix method is defined as follows:

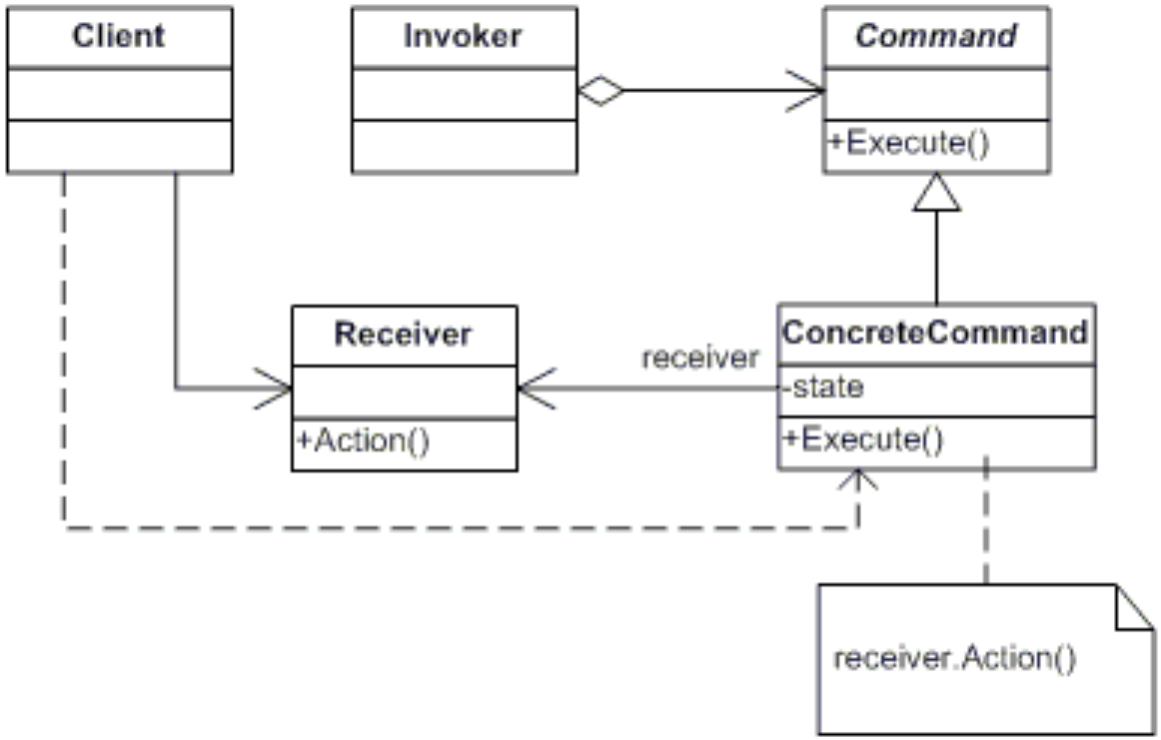
```
bool evaluate_postfix (Expr_Iterator & iter, int & result) {
  Stack <int> s;
  for (; !iter.is_done (); iter.advance ()) {
  switch (iter->tag_)
  case OPERATOR:
    int n1 = r.pop (), n2 = r.pop ();
    switch ((*iter).op_) {
    case '+': r.push (n2 + n1); break;
    case '-': r.push (n2 - n1); break;
    case '*': r.push (n2 * n1); break;
    case '/': r.push (n2 / n1); break;
    }
  case NUM:
    r.push (iter->num_); break;
  default:
    return false;
  }

  result = s.pop ();
  return true;
}
```

What happens when developers want to add new operators?

The Command Pattern

Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different request, queue or log requests, and support undoable operations



The Command Pattern - Example

Currently, the evaluate_postfix method is defined as follows:

```
bool evaluate_postfix (Expr_Iterator & iter, int & result) {
  Stack <int> s;
  for (; !iter.is_done (); iter.advance ()) {
  switch (iter->tag_)
  case OPERATOR:
    int n1 = r.pop (), n2 = r.pop ();
    switch ((*iter).op_) {
    case '+': r.push (n2 + n1); break;
    case '-': r.push (n2 - n1); break;
    case '*': r.push (n2 * n1); break;
    case '/': r.push (n2 / n1); break;
    }
  case NUM:
    r.push (iter->num_); break;
  default:
    return false;
  }

  result = s.pop ();
  return true;
}
```

We know that the evaluate_postfix is not a good solution. So, let's improve it!!

The Command Pattern - Example

Currently, the evaluate_postfix method is defined as follows:

```
bool evaluate_postfix (Expr_Iterator & iter, int & result) {
  Stack <int> s;
  for (; !iter.is_done (); iter.advance ()) {
  switch (iter->tag_)
  case OPERATOR:
    int n1 = r.pop (), n2 = r.pop ();
    switch ((*iter).op_) {
    case '+': r.push (n2 + n1); break;
    case '-': r.push (n2 - n1); break;
    case '*': r.push (n2 * n1); break;
    case '/': r.push (n2 / n1); break;
    }
  case NUM:
    r.push (iter->num_); break;
  default:
    return false;
  }

  result = s.pop ();
  return true;
}
```

Each of the operators take elements from the stack, and push a new element onto the stack

The Command Pattern - Example

Currently, the evaluate_postfix method is defined as follows:

```
bool evaluate_postfix (Expr_Iterator & iter, int & result) {
  Stack <int> s;
  for (; !iter.is_done (); iter.advance ()) {
  switch (iter->tag_)
  case OPERATOR:
    int n1 = r.pop (), n2 = r.pop ();
    switch ((*iter).op_) {
    case '+': r.push (n2 + n1); break;
    case '-': r.push (n2 - n1); break;
    case '*': r.push (n2 * n1); break;
    case '/': r.push (n2 / n1); break;
    }
  case NUM:
    r.push (iter->num_); break;
  default:
    return false;
  }

  result = s.pop ();
  return true;
}
```

For numbers, we are only pushing elements onto the stack.

The Command Pattern - Example

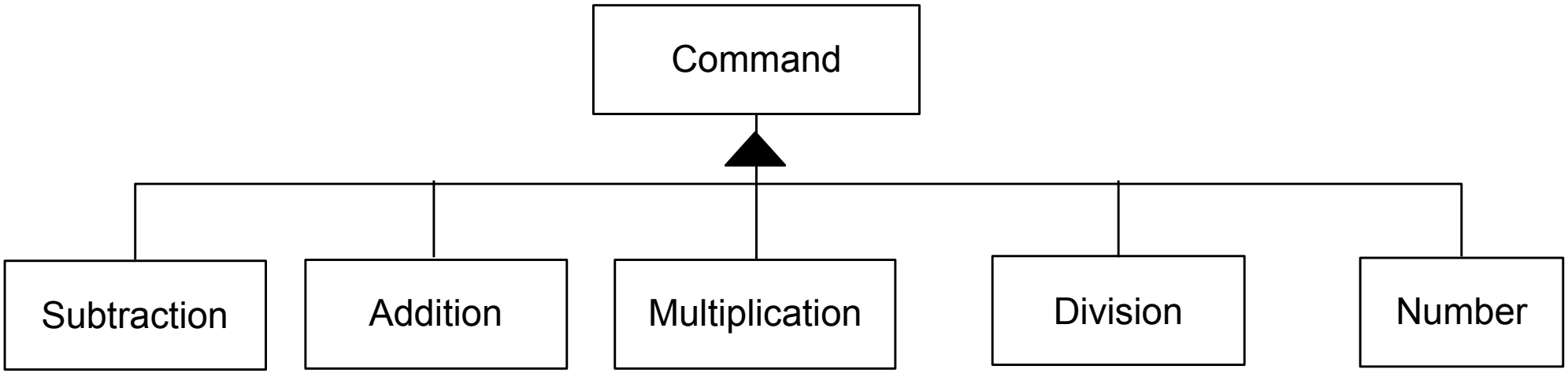
Currently, the evaluate_postfix method is defined as follows:

```
bool evaluate_postfix (Expr_Iterator & iter, int & result) {
  Stack <int> s;
  for (; !iter.is_done (); iter.advance ()) {
  switch (iter->tag_)
  case OPERATOR:
    int n1 = r.pop (), n2 = r.pop ();
    switch ((*iter).op_) {
    case '+': r.push (n2 + n1); break;
    case '-': r.push (n2 - n1); break;
    case '*': r.push (n2 * n1); break;
    case '/': r.push (n2 / n1); break;
    }
  case NUM:
    r.push (iter->num_); break;
  default:
    return false;
  }

  result = s.pop ();
  return true;
}
```

For all cases, we either pushing a number onto the stack, or removing numbers from the stack, e.g., only modifying the stack.

First Design of the Calculator

The Command Patterns can be used to capture the different operations (or Commands) the calculator must perform



The Command Pattern - Example

Since each operation is modifying the stack, we can assume each operation is a command that operates on the stack

```
class Expr_Command {
public:
  virtual void execute (void) = 0;
};
```

We can then define each operation in terms of Expr_Command

```
class Add_Command : public Expr_Command {
public:
  Add_Command (Stack <int> & s) : s_ (s) { }

  virtual void execute (void){
    int n2 = s_.pop (), n1 = s_.pop ();
    s_.push (n1 + n2);
  }

private:
  Stack <int> & s_;
};
```

The Command Pattern - Example

Since each operation is modifying the stack, we can assume each operation is a command that operates on the stack

```
class Expr_Command {
public:
  virtual void execute (void) = 0;
};
```

We can then define each operation in terms of Expr_Command

```
class Num_Command : public Expr_Command {
public:
  Num_Command (Stack <int> & s, int n)
    : s_ (s), n_ (n) { }

  void execute (void){
    s.push (this->n_);
  }

private:
  Stack <int> & s_;
  int n_;
};
```

The Command Pattern - Example

Instead of using an Expr_Iterator, we can now assume the array is a collection of Expr_Command elements, and we are using an Expr_Command_Iterator

```
void evaluate_postfix (Expr_Command_Iterator & iter) {
  for (; !iter.is_done (); iter.advance ())
    (*iter)->execute ();
}
```

Iterates over each Expr_Command element, and executes it command

The Command Pattern - Example

It is also possible to use the Template Method pattern for the all the Binary operators (e.g., +, -, *, /) to improve their implementation (and quality)

```
class Add_Command : public Expr_Command {
public:
  // ...
  virtual void execute (void){
    int n2 = s_.pop (), n1 = s_.pop ();
    s_.push (n1 + n2);
  }
};
```

```
class Subtract_Command : public Expr_Command {
public:
  // ...
  virtual void execute (void){
    int n2 = s_.pop (), n1 = s_.pop ();
    s_.push (n1 - n2);
  }
};
```

Binary operators always use the two top-most elements on the stack

The Command Pattern - Example

It is also possible to use the Template Method pattern for the all the Binary operators (e.g., +, -, *, /) to improve their implementation (and quality)

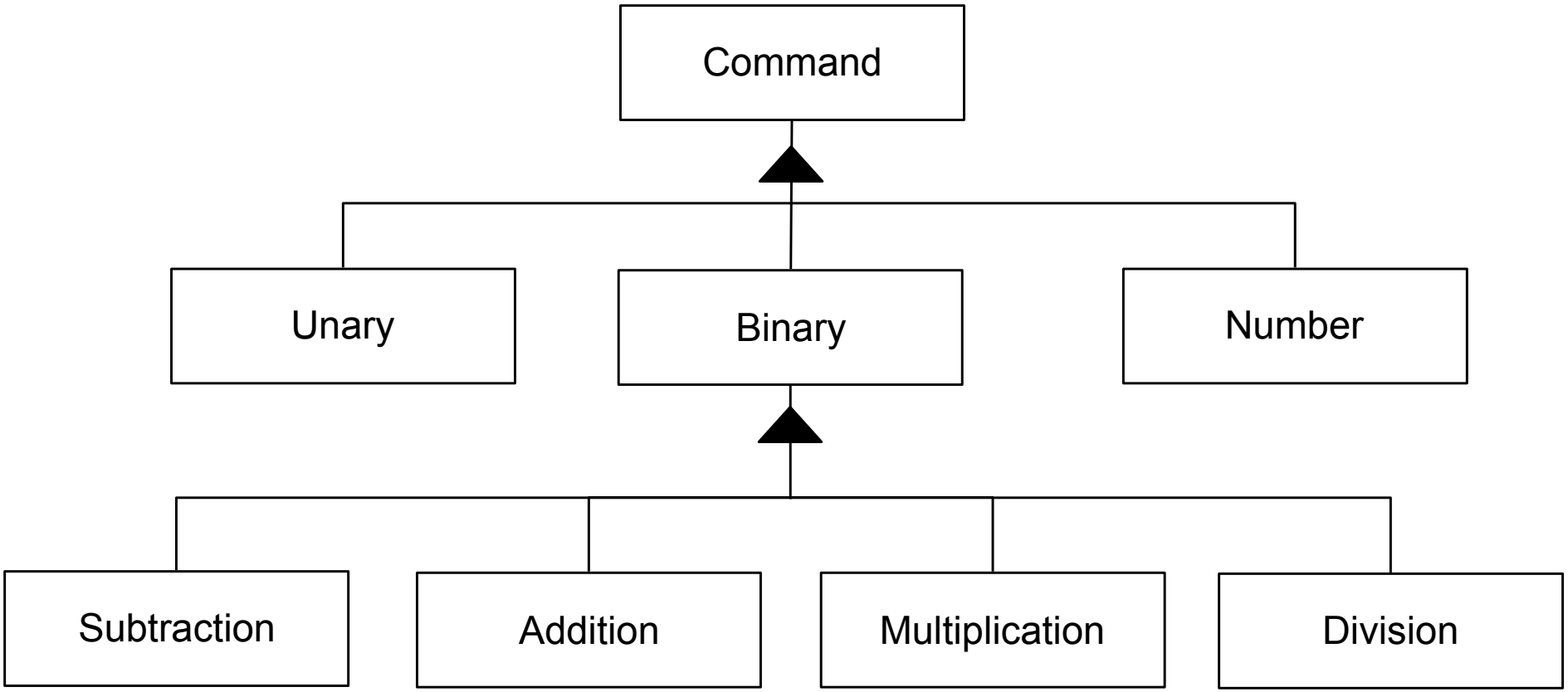
```
class Add_Command : public Expr_Command {
public:
  // ...
  virtual void execute (void){
    int n2 = s_.pop (), n1 = s_.pop ();
    s_.push (n1 + n2);
  }
};
```

The result of the binary operation is pushed onto the stack

```
class Subtract_Command : public Expr_Command {
public:
  // ...
  virtual void execute (void){
    int n2 = s_.pop (), n1 = s_.pop ();
    s_.push (n1 - n2);
  }
};
```

Re-Design of the Calculator

The Command Patterns can be used to capture the different operations (or Commands) the calculator must perform



The Command Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Binary_Op_Command : public Expr_Command {
public:
  bool execute (void) {
    int n2 = s_.pop (), n1 = s_.pop ();
    int result = this->evaluate (n1, n2);
    s_.push (result);
  }

  virtual int evaluate (int n1, int n2) const = 0;

  // ...
};
```

execute () removes the two top-most elements

The Command Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Binary_Op_Command : public Expr_Command {
protected:
  Binary_Op_Command (Stack <int> & s)
    : s_ (s){ }

public:
  bool execute (void) {
    int n2 = s_.pop (), n1 = s_.pop ();
    int result = this->evaluate (n1, n2);
    s_.push (result);
  }

  virtual int evaluate (int n1, int n2) const = 0;

private:
  Stack <int> &  s_;
  // ...
};
```

It passes the numbers to evaluate and gets the result

The Command Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Binary_Op_Command : public Expr_Command {
public:
  bool execute (void) {
    int n2 = s_.pop (), n1 = s_.pop ();
    int result = this->evaluate (n1, n2);
    s_.push (result);
  }

  virtual int e                          n2) const = 0;

  // ...
};
```

The result is pushed back onto the stack.

The Command Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Binary_Op_Command : public Expr_Command {
public:
  bool execute (void) {
    int n2 = s_.pop (), n1 = s_.pop ();
    int result = this->evaluate (n1, n2);
    s_.push (result);
  }

  virtual int evaluate (int n1, int n2) const = 0;

  // ...
};
```

evaluate () is defined as pure virtual, which requires subclasses to implement it

The Command Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

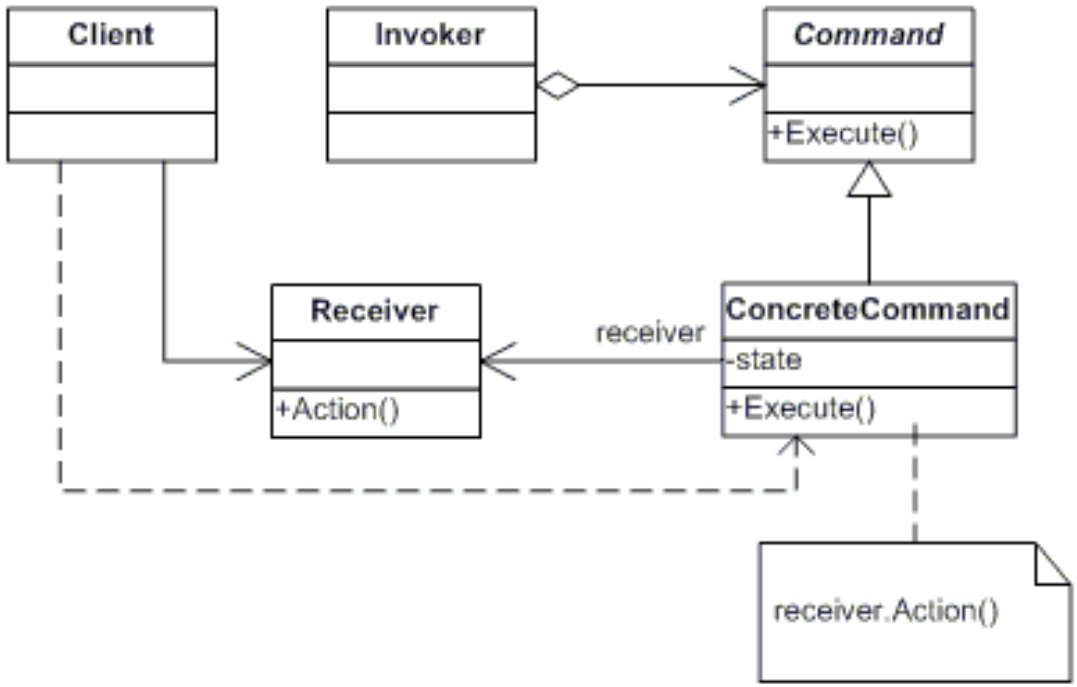
```
class Binary_Op_Command : public Expr_Command {
public:
  bool execute (void) {
    int n2 = s_.pop (), n1 = s_.pop ();
    int result = this->evaluate (n1, n2);
    s_.push (result);
  }

  virtual int evaluate (int n1, int n2) const

  // ...
};

class Add_Command : public Binary_Op_Command {
  int evaluate (int n1, int n2){
    return n1 + n2;
  }
}
```

When we define binary operators, we only have to define the evaluate () function

The Command Pattern - Consequences

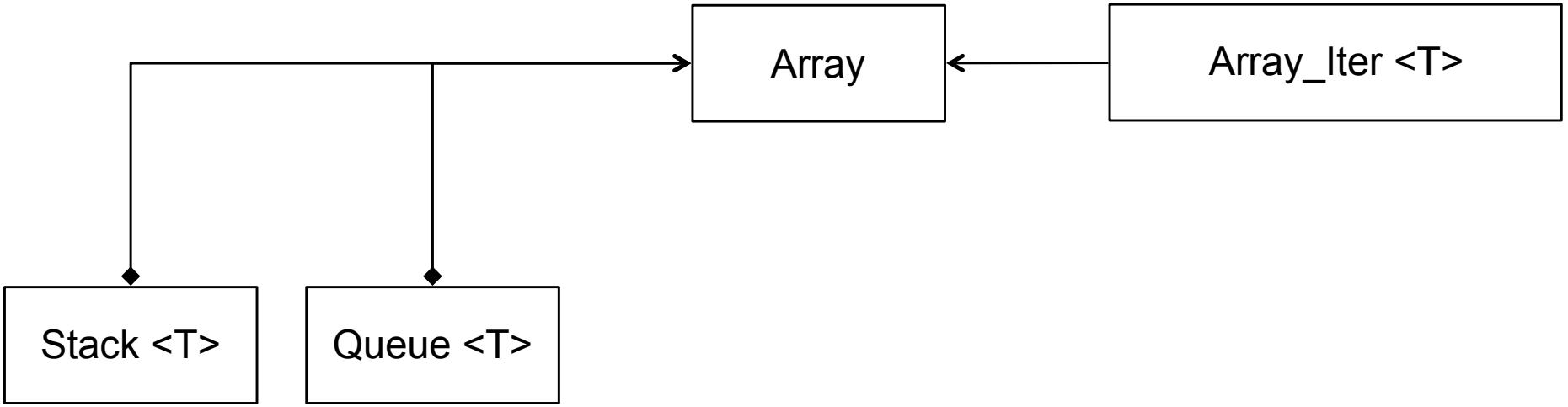
The Command Patterns has the following consequences:

1. Command decouples the object that invokes the operation from the one that knows how to perform it
2. Commands are first-class objects
3. You can assemble (or compose) commands to create composite commands
4. Its easy to add new commands because you do not have the changes existing classes
5. Adds a level of indirection



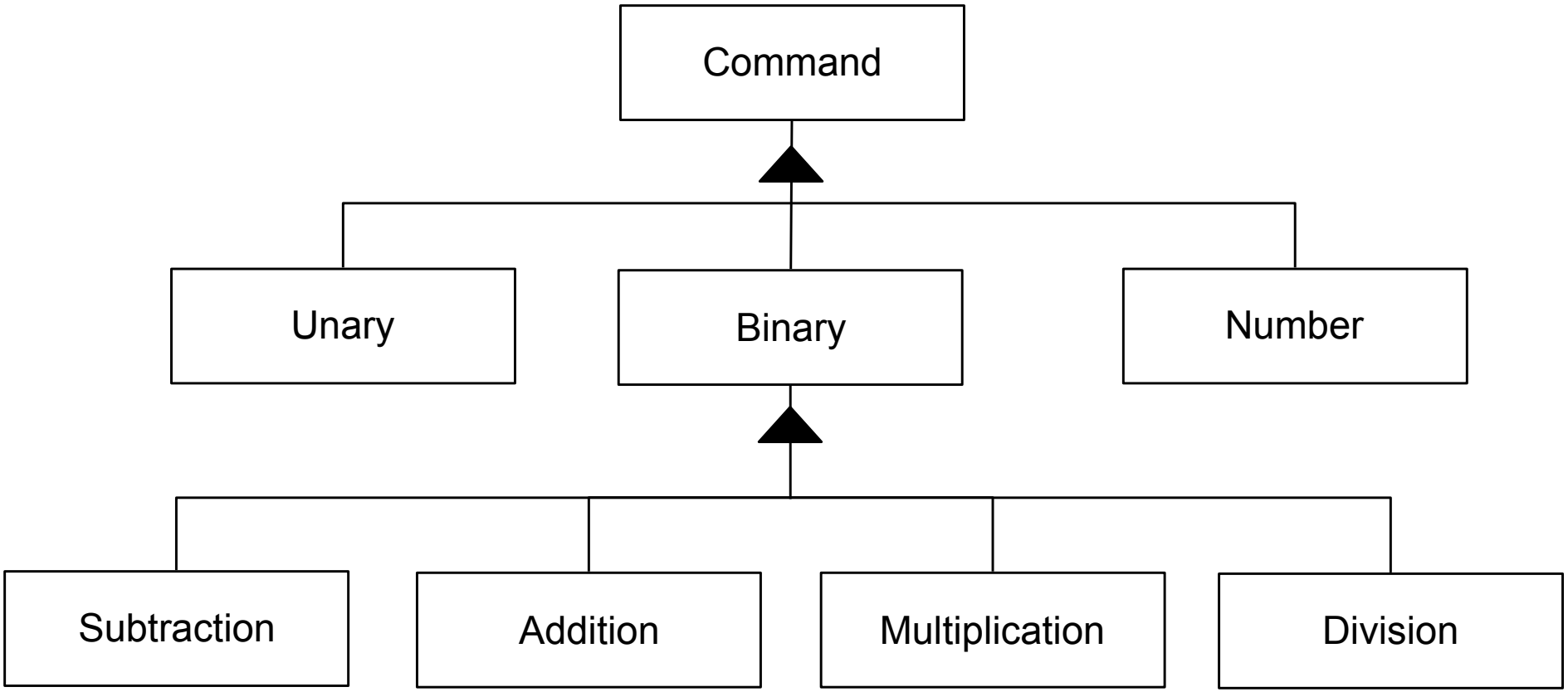
Current Design of the Calculator

So far, we have used design patterns to define the low-level abstraction (or objects) of the calculator



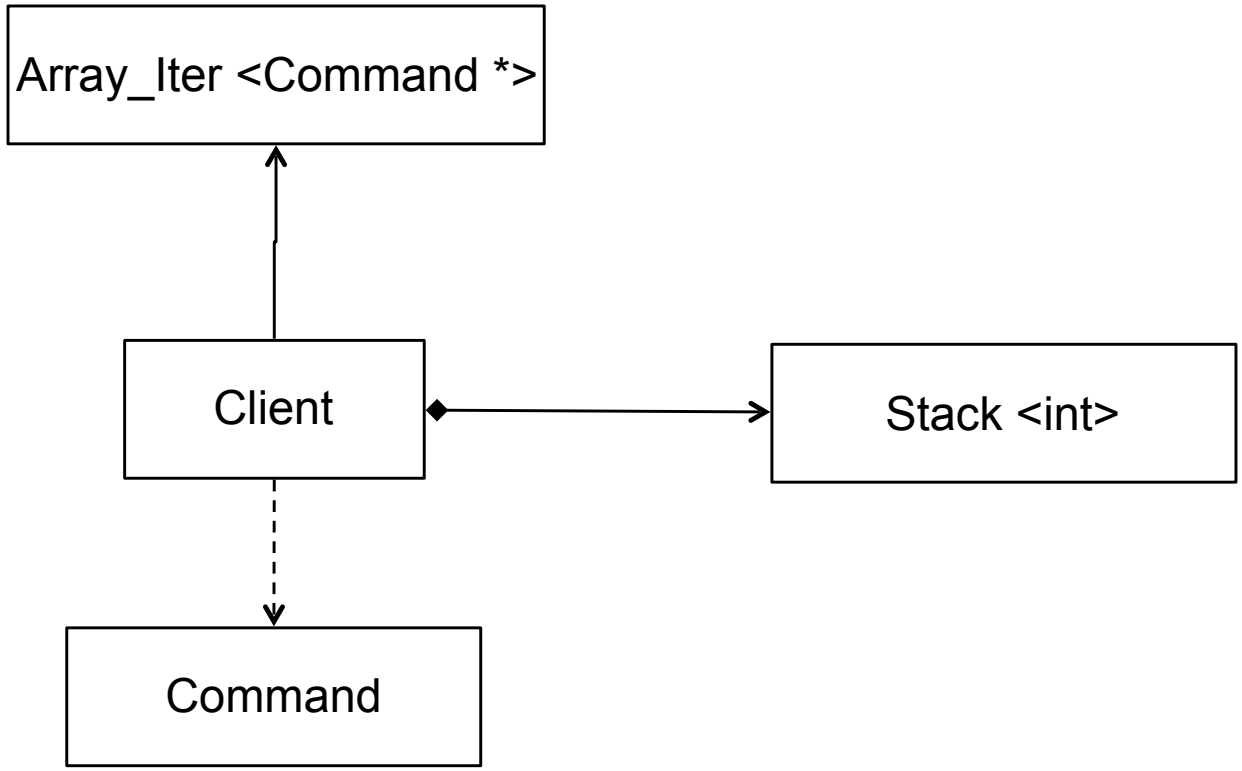
Current Design of the Calculator

The Command Patterns can be used to capture the different operations (or Commands) the calculator must perform



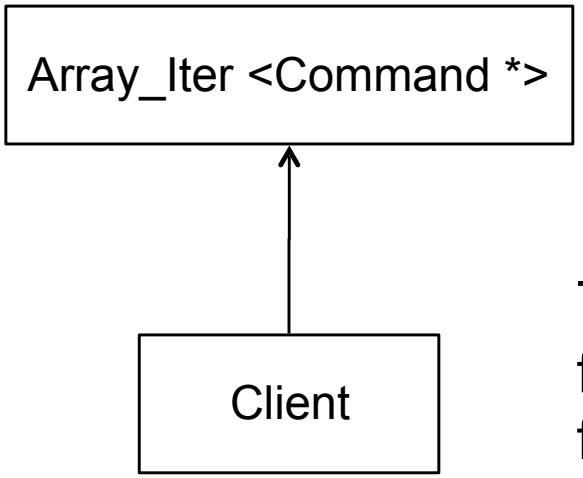
Current Design of the Calculator

We now have taken the low-level abstractions (along with other software design patterns) to realize the client of the calculator



Current Design of the Calculator

We know that Array_Iter allows us to iterate over the Array of Command elements, and evaluate expression in O(n) time



The Command objects in Array are in postfix format, but the expression is originally in infix format. How to we get from Infix to Postfix format?

The Abstract Factory Pattern – Motivation

Have you ever been in the following situation:

- Have a set of closely related objects that need to be created, but do not care on their concrete type
    - e.g., just need to create new objects
- Have a set of closely related objects that are in different families (or groupings)
    - e.g., the same (object) part(s) for different versions of a car ADT

Pattern Analogy

How does a parts plant produce different versions of a car part without having to change its entire manufacturing operation for each part?

The Abstract Factory Pattern – Motivation Ex.

We know that the expression the calculator evaluates is in infix format

5 + 4

We know that calculator is expected to evaluate the postfix version of the expression

5 4 +

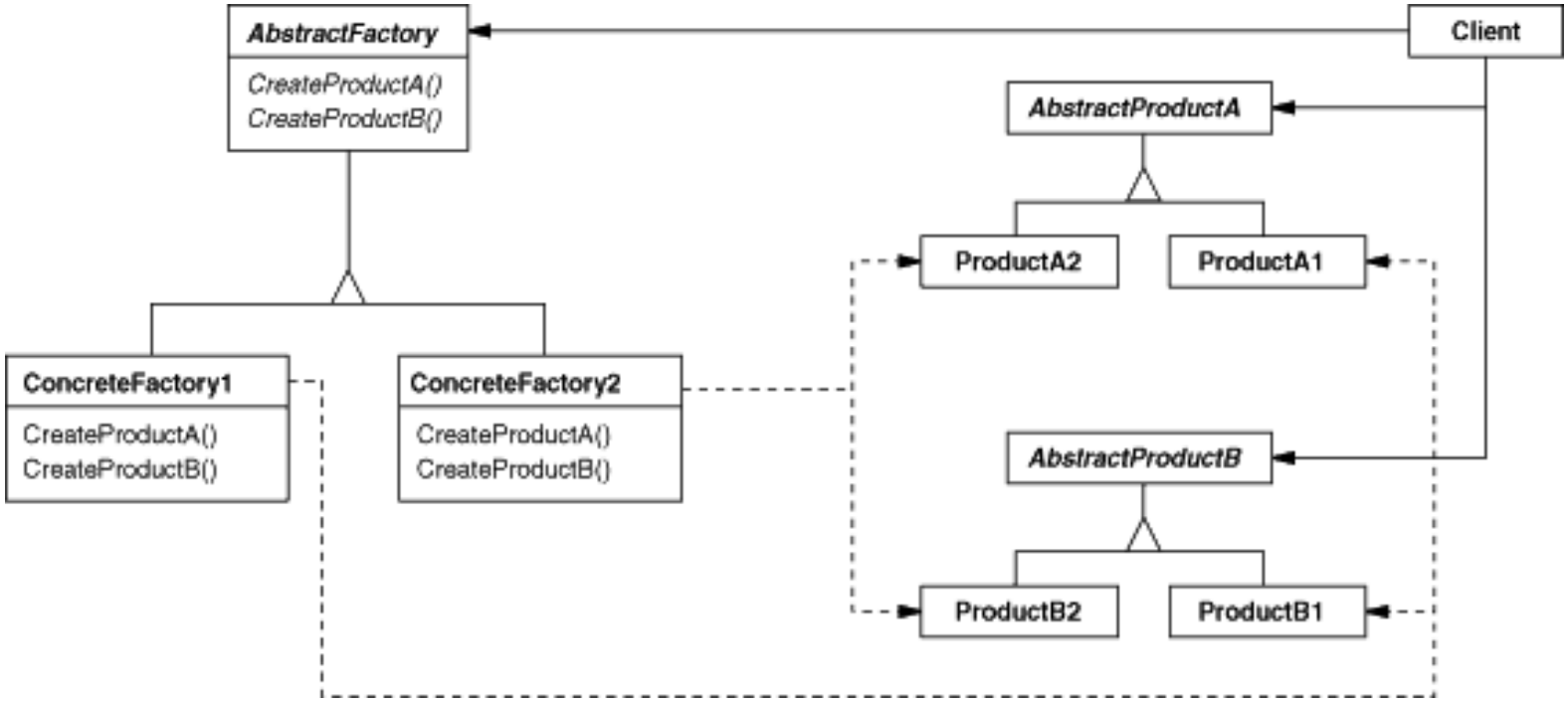
Finally, we know that each “entity” in the expression can be represented as a command

We just need to figure how to convert the infix expression (string) into a postfix expression (an array of command elements in postfix order)

The Abstract Factory Pattern

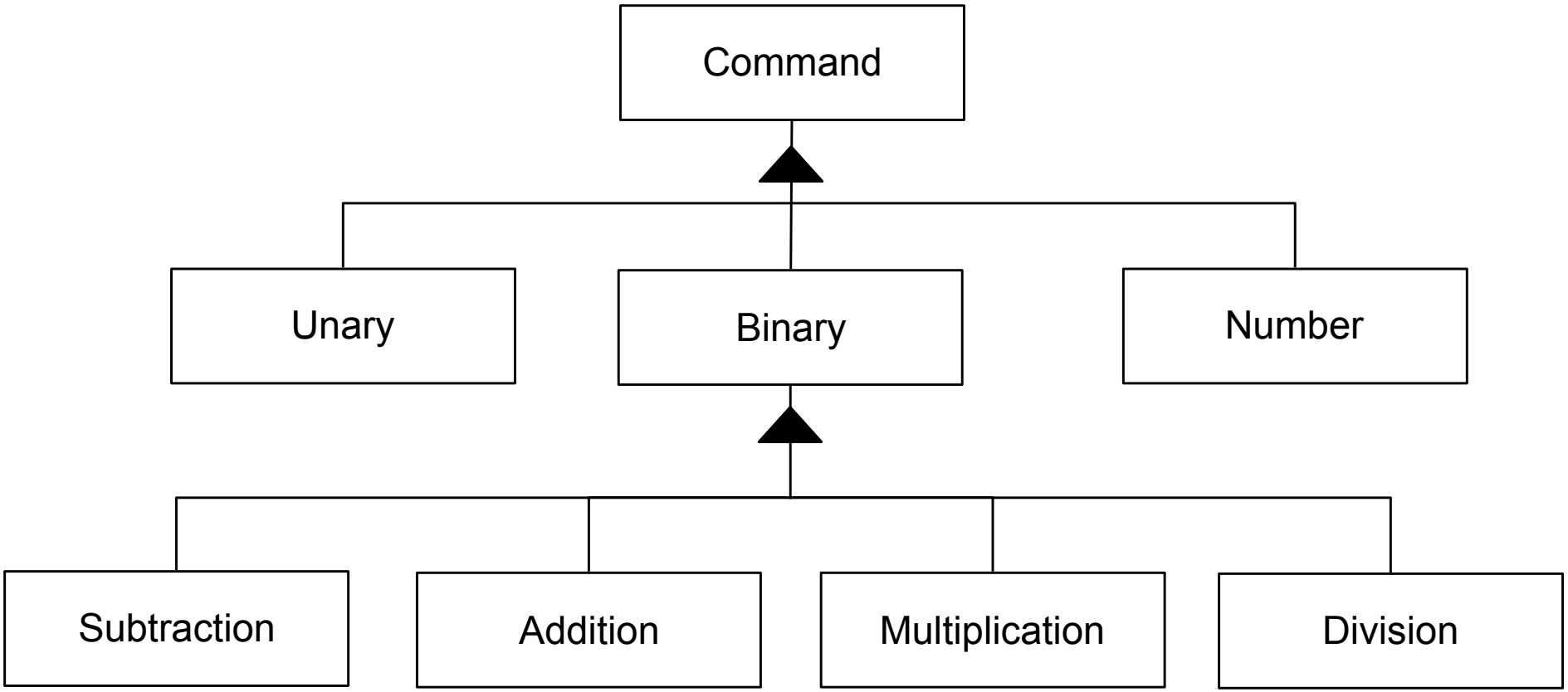
Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes



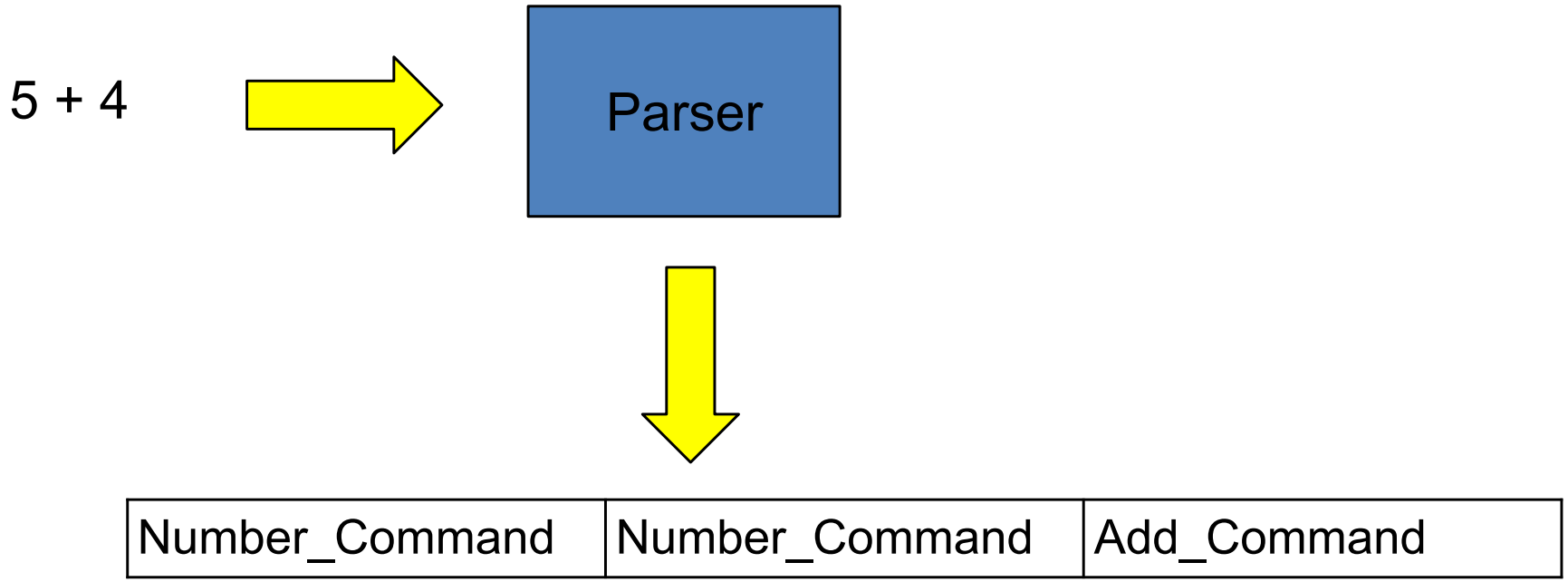
The Abstract Factory Pattern – Example

Each entity in the expression can be represented as a command object that will eventually appear in the postfix version of the expression



The Abstract Factory Pattern – Example

Since the expression is given in infix format, the goal is to parse the expression an “create” the appropriate Command object for each entity that is parsed



Each element in the Array is the result of a “create” operation

The Abstract Factory Pattern – Example

Our knowledge of the target solution allows us to defined the following interface for creating objects:

```
class Expr_Command_Factory
{
public:
  virtual ~Expr_Command_Factory (void) = 0;

  virtual Number_Command * create_number_command (int num) = 0;

  virtual Add_Command * create_add_command (void) = 0;

  virtual Subtract_Command * create_subtract_command (void) = 0;

  // ...
private:
  // prevent the following operations
  Expr_Command_Factory (const Expr_Command_Factory &);
  const Expr_Command_Factory & operator = (const Expr_Command_Factory &);
};
```

The Abstract Factory Pattern – Example

Our knowledge of the target solution allows us to defined the following interface for creating objects:

```
class Expr_Command_Factory
{
public:
  virtual ~Expr_Command_Factory (void) = 0;

  virtual Number_Command * create_number_command (int num) = 0;

  virtual Add_Command * create_add_command (void) = 0;

  virtual Subtract_Command * create_subtract_command (void) = 0;

  // ...
private:
  // prevent the following operations
  Expr_Command_Factory (const Expr_Command_Factory &);
  const Expr_Command_Factory & operator = (const Expr_Command_Factory &);
};
```

Pure virtual methods make this an pure abstract class (or interface); subclasses must override each method

The Abstract Factory Pattern – Example

Our knowledge of the target solution allows us to defined the following interface for creating objects:

```
class Expr_Command_Factory
{
public:
  virtual ~Expr_Command_Factory (void) = 0;

  virtual Number_Command * create_number_command (int num) = 0;

  virtual Add_Command * create_add_comma

  virtual Subtract_Command * create_subt

  // ...
private:
  // prevent the following operations
  Expr_Command_Factory (const Expr_Command_Factory &);
  const Expr_Command_Factory & operator = (const Expr_Command_Factory &);
};
```

Prevent the client from calling the following operations in all subclasses

The Abstract Factory Pattern – Example

We use the Expr_Command_Factory by deriving a concrete from it that will be responsible for creating the “concrete” objects:

```
class Stack_Expr_Command_Factory : public Expr_Command_Factory
{
public:
  Stack_Expr_Command_Factory (Stack <int> & stack);

  virtual Number_Command * create_number_command (int num);

  virtual Add_Command * create_add_command (void);

  virtual Subtract_Command * create_subtract_command (void);

private:
  Stack <int> & stack_;
};
```

In the subclass, each method is implemented and returns the concrete object

The Abstract Factory Pattern – Example

Finally, we can use the Expr_Command_Factory to create Command objects for each entity when parsing the infix expression and converting it to a postfix expression:

```
bool infix_to_postfix (const std::string & infix,
                       Expr_Command_Factory & factory,
                       Array <Command *> & postfix) {
  std::istringstream input(infix); // create a input stream parser
  std::string token;               // current token in string/stream
  Command * cmd = 0;               // created command object
  Stack <Command *> temp;
  while (!input.eof ()) {
     input >> token;
     if (token == "+")
       cmd = factory.create_add_command ();
     else if (token == "-")
       cmd = factory.create_subtract_command ();
     // ...
     // handle the command based on infix-to-postfix algorithm
  }
  return true;
};
```

The Abstract Factory Pattern – Example

Finally, we can use the Expr_Command_Factory to create Command objects for each entity when parsing the infix expression and converting it to a postfix expression:

```
bool infix_to_postfix (const std::string & infix,
                       Expr_Command_Factory & factory,
                       Array <Command *> & postfix) {
  std::istringstream input(infix); // create a input stream parser
  std::string token;               // current token in string/stream
  Command * cmd = 0;               // created command object
  Stack <Command *> t
  while (!input.eof
    input >>
    if (token
      cmd = f
    else if (token == "-")
      cmd = factory.create_subtract_command ();
    // ...
    // handle the command based on infix-to-postfix algorithm
  }
  return true;
};
```

The Expr_Command_Factory for creating the Command objects

The Abstract Factory Pattern – Example

Finally, we can use the Expr_Command_Factory to create Command objects for each entity when parsing the infix expression and converting it to a postfix expression:

```
bool infix_to_postfix (const std::string & infix,
                       Expr_Command_Factory & factory,
                       Array <Command *> & postfix) {
  std::istringstream input(infix); // create a input stream parser
  std::string token;               // current token in string/stream
  Command * cmd = 0;               // created command object
  Stack <Command *> temp;
  while (!input.eof ()) {
     input >> token;
     if (token == "+")
       cmd = factory.cre
     else if (token == "
       cmd = factory.crea
     // ...
     // handle the command based on infix-to-postfix algorithm
  }
  return true;
};
```

Stream for parsing the tokens in the expression

The Abstract Factory Pattern – Example

Finally, we can use the Expr_Command_Factory to create Command objects for each entity when parsing the infix expression and converting it to a postfix expression:

```
bool infix_to_postfix (const std::string & infix,
                       Expr_Command_Factory & factory,
                       Array <Command *> & postfix) {
  std::istringstream input(infix); // create a input stream parser
  std::string token;               // current token in string/stream
  Command * cmd = 0;               // created command object
  Stack <Command *> temp;
  while (!input.eof ()) {
    input >> token;
    if (token == "+")
      cmd = factory.create_plus_command ();
    else if (token == "-")
      cmd = factory.create_minus_command ();
    // ...
    // handle the command in the infix to postfix algorithm
  }
  return true;
};
```

Stack for assisting with infix to postfix conversion

The Abstract Factory Pattern – Example

Finally, we can use the Expr_Command_Factory to create Command objects for each entity when parsing the infix expression and converting it to a postfix expression:

```
bool infix_to_postfix (const std::string & infix,
                       Expr_Command_Factory & factory,
                       Array <Command *> & postfix) {
  std::istringstream input(infix); // create a input stream parser
  std::string token;               // current token in string/stream
  Command * cmd = 0;               // created command object
  Stack <Command *> temp;
  while (!input.eof ()) {
     input >> token;
     if (token == "+")
       cmd = factory.create_add_command ();
     else if (token == "-")
       cmd = factory.create_subtract_command ();
     // ...
     // handle the command based on infix-to-postfix algorithm
  }
  return true;
};
```

Logic for creating the Command objects

The Abstract Factory Pattern – Example

Finally, we can use the Expr_Command_Factory to create Command objects for each entity when parsing the infix expression and converting it to a postfix expression:

```
bool infix_to_postfix (const std::string & infix,
                       Expr_Command_Factory & factory,
                       Array <Command *> & postfix) {
  std::istringstream input(infix); // create a input stream parser
  std::string token;               // current token in string/stream
  Command * cmd = 0;               // created command object
  Stack <Command *> temp;
  while (!input.eof ()) {
     input >> token;
     if (token == "+")
       cmd = factory.create_add_command ();
     else if (token == "-")
       cmd = factory.create_subtract_command ();
     // ...
     // handle the command based on infix-to-postfix algorithm
  }
  return true;
};
```

Logic for handling the command

The Abstract Factory Pattern – Example

The client uses the infix_to_postfix conversion algorithm as follows:

```
// get input from STDIN
std::string infix;

// ...
Stack <int> result;
Stack_Expr_Command_Factory factory (result);

Array <Command *> postfix;
infix_to_postfix (infix, factory, postfix);

// evaluate postfix

int res = result.top ();
```

Declare the concrete factory

Invoke the infix_to_postfix () function

The Abstract Factory Pattern - Consequences

The Abstract Factory Pattern has the following consequences:

- Isolates concrete classes
  - i.e., lets you *control* the concrete classes of objects that an applications creates

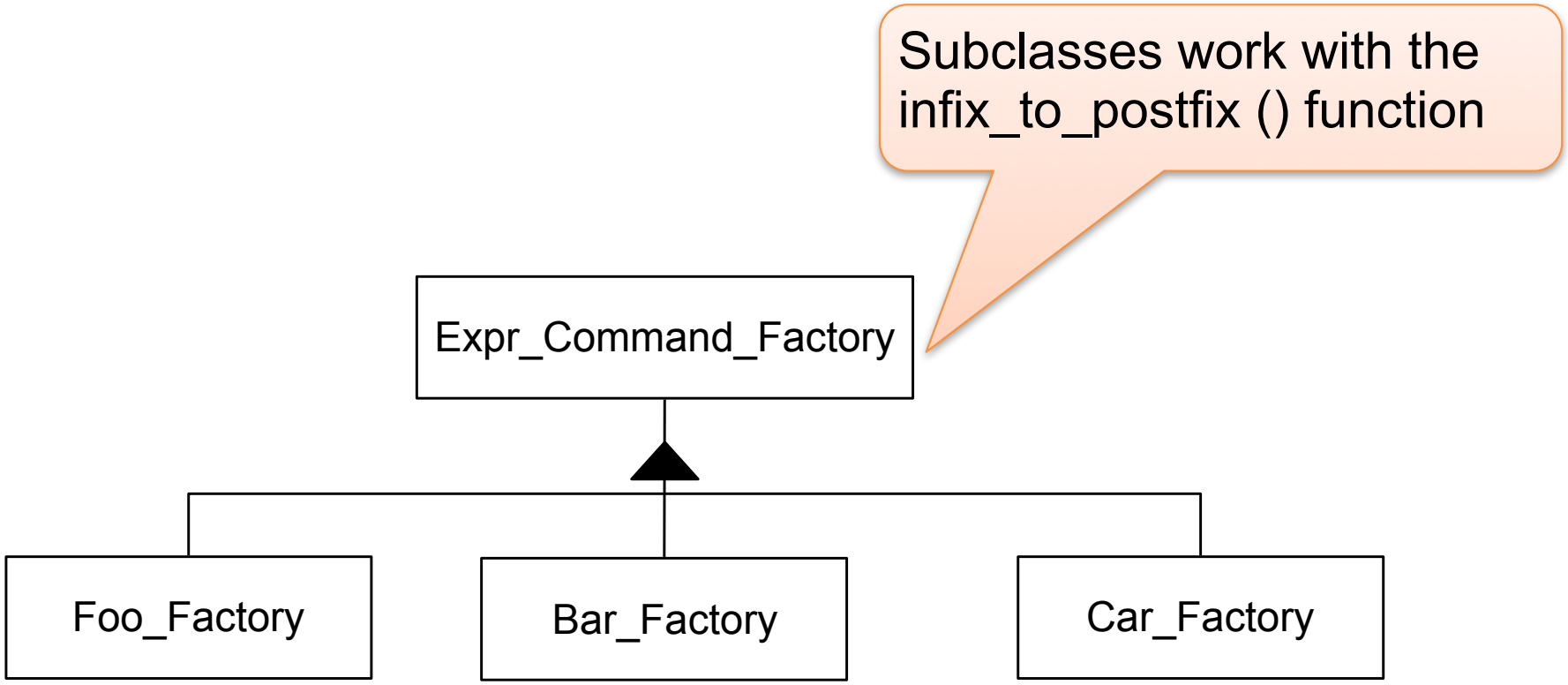
Each method control what class/object is created

```
class Expr_Command_Factory
{
public:
  virtual ~Expr_Command_Factory (void) = 0;
  virtual Number_Command * create_number_command (int num) = 0;
  virtual Add_Command * create_add_command (void) = 0;
  virtual Subtract_Command * create_subtract_command (void) = 0;

  // …
};
```

The Abstract Factory Pattern - Consequences

The Abstract Factory Pattern has the following consequences:

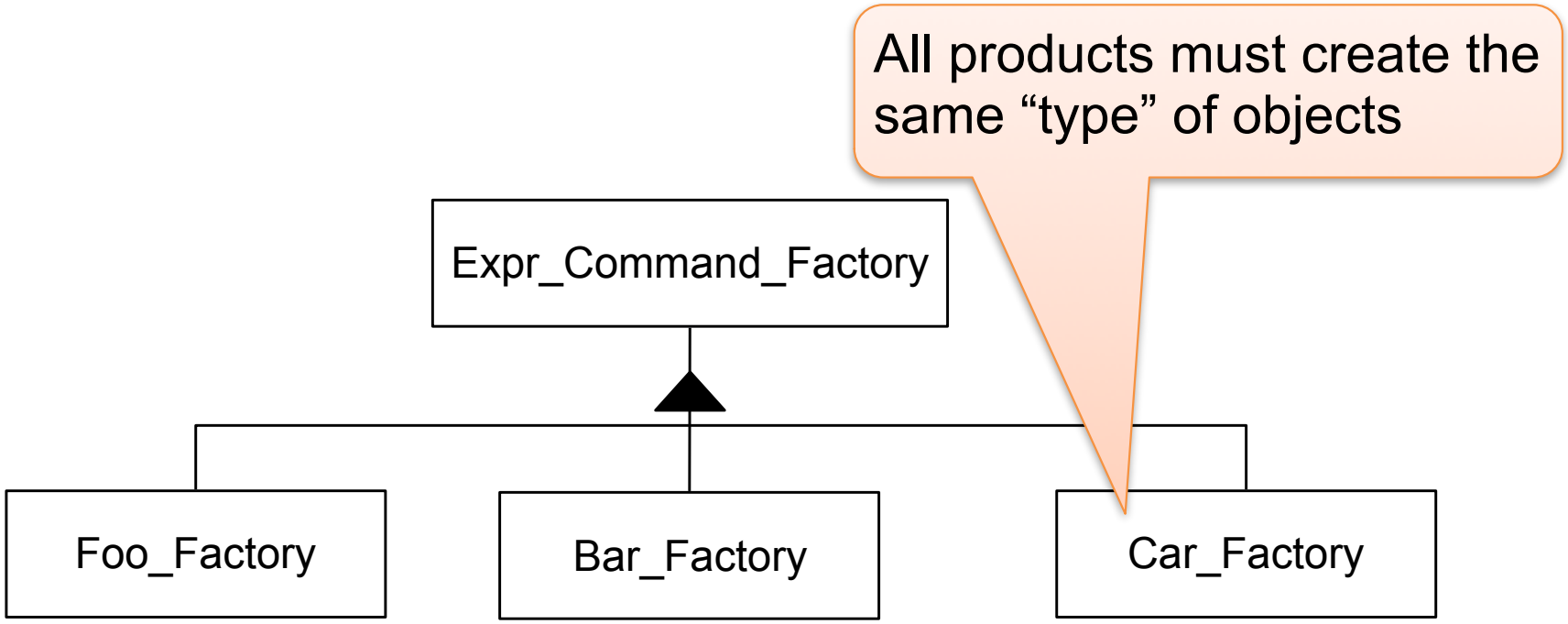
- Isolates concrete classes
  - i.e., lets you *control* the concrete classes of objects that an applications creates
- Makes interchanging product families easy



The Abstract Factory Pattern - Consequences

The Abstract Factory Pattern has the following consequences:

- Isolates concrete classes
  - i.e., lets you *control* the concrete classes of objects that an applications creates
- Makes interchanging product families easy
- Promotes consistency among products



The Abstract Factory Pattern - Consequences

The Abstract Factory Pattern has the following consequences:

- Isolates concrete classes
  - i.e., lets you *control* the concrete classes of objects that an applications creates
- Makes interchanging product families easy
- Promotes consistency among products
- Supporting new kinds of products is difficult
  - e.g., new factories must implement all the factory methods & supported elements are already defined

```
class Expr_Command_Factory
{
public:
  virtual ~Expr_Command_Factory (void) = 0;
  virtual Number_Command * create_number_command (int num) = 0;
  virtual Add_Command * create_add_command (void) = 0;
  virtual Subtract_Command * create_subtract_command (void) = 0;

  // …
};
```

Must implement all & controls object type

Our Command Abstract Factory

Our knowledge of the target solution allows us to define the following interface for creating commands:

```
class Expr_Command_Factory
{
public:
  virtual ~Expr_Command_Factory (void) = 0;

  virtual Number_Command * create_number_command (int num) = 0;

  virtual Add_Command * create_add_command (void) = 0;

  virtual Subtract_Command * create_subtract_command (void) = 0;

  // ...
private:
  // prevent the following operations
  Expr_Command_Factory (const Expr_Command_Factory &);
  const Expr_Command_Factory & operator = (const Expr_Command_Factory &);
};
```

Our Stack Expression Command Factory

We use the Expr_Command_Factory by deriving a concrete from it that will be responsible for creating the “concrete” objects:

```
class Stack_Expr_Command_Factory : public Expr_Command_Factory
{
public:
  Stack_Expr_Command_Factory (Stack <int> & stack);

  virtual Number_Command * create_number_command (int num);

  virtual Add_Command * create_add_command (void);

  virtual Subtract_Command * create_subtract_command (void);

private:
  Stack <int> & stack_;
};
```

In the subclass, each method is implemented and returns the concrete object

Our Expression Commands

Since each operation is modifying the stack, we can assume each operation is a command that operates on the stack

```
class Expr_Command {
public:
  virtual void execute (void) = 0;

protected:
  Stack <int> & s_;
};
```

We can then define each operation in terms of Expr_Command

```
class Add_Command : public Expr_Command {
public:
  Add_Command (Stack <int> & s) : Expr_Command (s) { }

  virtual void execute (void){
    int n2 = s_.pop (), n1 = s_.pop ();
    s_.push (n1 + n2);
  }
};
```

Our Expression Commands - Redesign

In an alternative design, we decide to pass the target stack to the execute command instead of storing it as a reference:

```
class Expr_Command {
public:
  virtual void execute (Stack <int> & s) = 0;
};
```

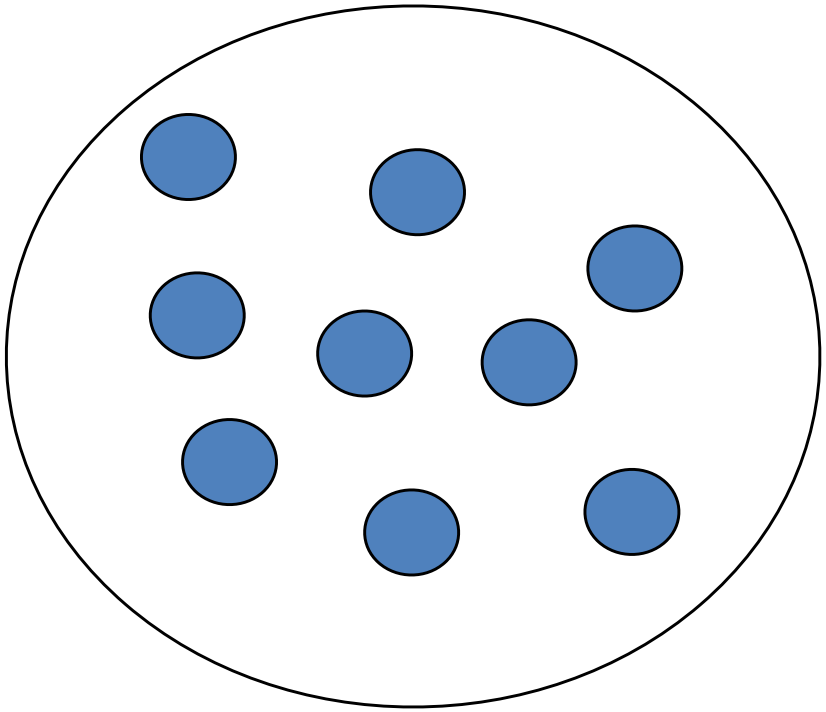
We can then define each operation in terms of Expr_Command

```
class Add_Command : public Expr_Command {
public:
  Add_Command (void) { }

  virtual void execute (Stack <int> & s){
    int n2 = s.pop (), n1 = s.pop ();
    s.push (n1 + n2);
  }
};
```

The Flyweight Pattern – Motivation

Have you ever been in the following situation:

- Realize that your application needs many objects, but a naïve implementation would have a negative impact on performance
  - *e.g.*, not scale, require too much memory
- Have many objects, but the same object can operate in multiple contexts simultaneously
- Have an object that is stateless, and its behavior is determined by its operations on an external context



Unique Object Instances

The Flyweight Pattern – Motivating Ex.

In the postfix representation of the expression evaluator, we did the following:

1. Converted an infix expression to a postfix expression where each item in the postfix expression was its own command object

Postfix Representation

-5	3	4	+	*

2. Defined a stack outside of postfix expression that managed current state
3. Iterated over each item in postfix expression for evaluation

```
void evaluate_postfix (Expr_Command_Iterator & iter) {
  for (; !iter.is_done (); iter.advance ())
   (*iter)->execute ()
}
```

The Flyweight Pattern – Motivating Ex.

In the postfix representation of the expression evaluator, we did the following:

1. Converted an infix expression to a postfix expression where each item in the postfix expression was its own command object

Postfix Representation

-5	3	4	+	*

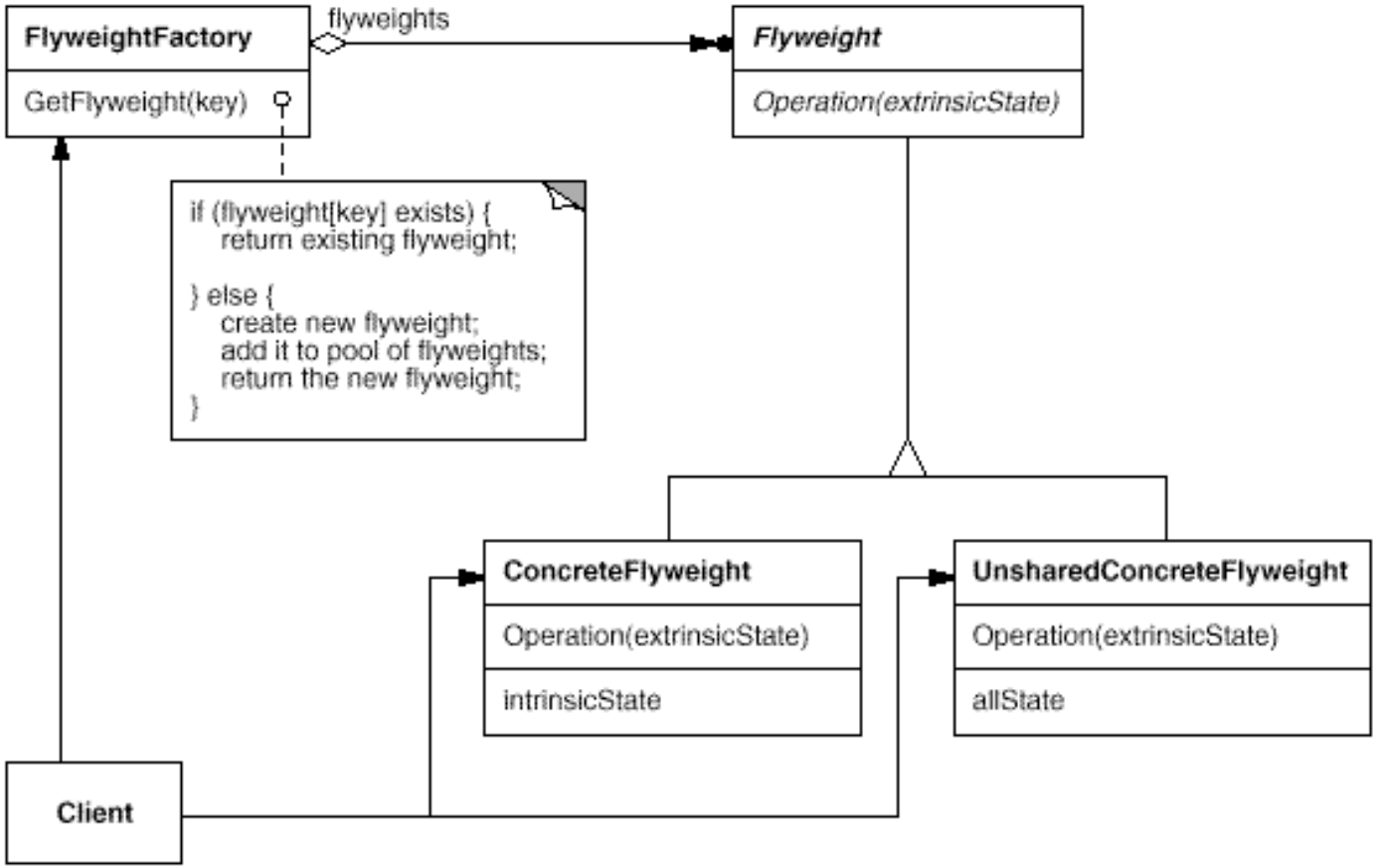
2. Defined a stack outside of postfix expression that managed current state
3. Iterated over each item in postfix expression for evaluation

```
void evaluate_postfix (Expr_Command_Iterator & iter) {
  for (; !iter.is_done (); iter.advance ())
   (*iter)->execute ()
}
```

In reality, many Commands in the array are “stateless”. How can we leverage this domain knowledge to reduce object instances & improve performance?

The Flyweight Pattern

Intent
Use sharing to support large numbers of fine-grained objects efficiently.



The Flyweight Pattern - Example

In the original postfix version of the expression factory, we used an abstract factory to create each of the command objects:

- e.g., addition, subtraction, multiplication, division, & modulus

```
bool infix_to_postfix (const std::string & infix,
                       Expr_Command_Factory & factory,
                       Array <  *> & postfix) {
  std::istringstream input(infix); // create a input stream parser
  std::string token;               // current  n in string/stream
  Command * cmd = 0;               // created  d object
  Stack <Command *> temp;
  while (!input.eof ()) {
     input >> token;
     if (token == "+")
       cmd = factory.create_add_command ();
     else if (token == "-")
       cmd = factory.create_subtract_command ();
     // ...
     // handle the command based on infix-to-postfix algorithm
  }
  return true;
};
```

Abstract factory responsible for creating commands

The Flyweight Pattern - Example

In the original postfix version of the expression factory, we used an abstract factory to create each of the command objects:

- e.g., addition, subtraction, multiplication, division, & modulus

```
bool infix_to_postfix (const std::string & infix,
                       Expr_Command_Factory & factory,
                       Array <Command *> & postfix) {
  std::istringstream input(infix); // create a input stream parser
  std::string token;               // current token in string/stream
  Command * cmd = 0;               // created command object
  Stack <Command *> temp;
  while (!input.eof ()) {
     input >> token;
     if (token == "+")
       cmd = factory.create_add_command ();
     else if (token == "-")
       cmd = factory.create_subtract_command ();
     // ...
     // handle the command based on infix-to-postfix algorithm
  }
  return true;
};
```

Usage of the abstract factory to create commands

The Flyweight Pattern - Example

The original concrete abstract factory created a new concrete command each time it was instructed to do so by the client

```
class Stack_Expr_Command_Factory : public Expr_Command_Factory
{
public:
  Stack_Expr_Command_Factory (Stack <int> & s)
    : stack_ (s) { }

  virtual Number_Command * create_number_command (int num) {
    return new Stack_Number_Command (this->stack_, num);
  }

  virtual Add_Command * create_add_command (void) {
    return new Stack_Add_Command (this->stack_);
  }

  virtual Subtract_Command * create_subtract_com    (void) {
    return new Stack_Subtract_Command (this->sta
  }
};
```

Each factory method on the abstract factory creates a new object

The Flyweight Pattern - Example

Deeper investigation of each command type for the postfix version of the expression evaluate reveals that operators are stateless:

```
class Binary_Op_Command {
public:
  Binary_Op_Command (Stack <int> & s)
    : s_ (s) { }

  virtual void execute (void){
    int n2 = s.pop (), n1 = s.pop ();
    this->s_.push (this->evaluate (n1, n2));
  }
}

class Stack_Add_Command : public Binary_Op_Command {
public:
  virtual bool evaluate (int n1, int n2){
    return n1 + n2;
  }
}
```

The Stack_Add_Command contains a Stack reference, but its execution state comes from the Stack.

The Flyweight Pattern - Example

The number command is the only command object that is considered stateful:

```
class Stack_Number_Command : public Number_Command {
public:
  Stack_Number_Command (Stack <int> & s, int num)
    : s_ (s), num_ (num) { }

  virtual void execute (void){
    this->s_.push (this->num_);
  }

private:
  Stack <int> & s_;
  int num_;
}
```

The Stack_Number_Command execution state is bound at creation time.

The Flyweight Pattern - Example

We can leverage the Flyweight pattern by implementing a new abstract factory that uses a single object for operators, and many objects for operands.

```
class Flyweight_Expr_Command_Factory : public Expr_Command_Factory
{
public:
  Flyweight_Expr_Command_Factory (Stack <int> & s)
    :s_ (s) { }

  virtual Number_Command * create_number_command (int num) {
    return new Stack_Number_Command (this->s_, num);
  }

  virtual Add_Command * crea     add_command (void) {
    return this->add_;
  }

  // ...

private:
  Stack_Add_Command * add_;
  // ...
};
```

Each number gets its own command object.

The Flyweight Pattern - Example

We can leverage the Flyweight pattern by implementing a new abstract factory that uses a single object for operators, and many objects for operands.

```
class Flyweight_Expr_Command_Factory : public Expr_Command_Factory
{
public:
  Flyweight_Expr_Command_Factory (Stack <int> & s)
    :s_ (s) { }

  virtual Number_Command * create_number_command (int num) {
    return new Stack_Number_Command (this->s_, num);
  }

  virtual Add_Command * create_add_command (void) {
    return this->add_;
  }

  // ...

private:
  Stack_Add_Command * add_;
  // ...
};
```

We create a single command for each operator.

The Flyweight Pattern - Example

We can leverage the Flyweight pattern by implementing a new abstract factory that uses a single object for operators, and many objects for operands.

```
class Flyweight_Expr_Command_Factory : public Expr_Command_Factory
{
public:
  Flyweight_Expr_Command_Factory (Stack <int> & s)
    :s_ (s) { }

  virtual Number_Command * create_number_command (int num) {
    return new Stack_Number_Command (this->s_, num);
  }

  virtual Add_Command * create_add_command (void) {
    return this->add_;
  }

  // ...

private:
  Stack_Add_Command * add_;
  // ...
};
```

We reuse the command object instead of always creating a new one.

The Flyweight Pattern - Example

We can leverage the Flyweight pattern by implementing a new abstract factory that uses a single object for operators, and many objects for operands.

```
class Flyweight_Expr_Command_Factory : public Expr_Command_Factory
{
public:
  Flyweight_Expr_Command_Factory (Stack <int> & s)
    :s_ (s) { }

  virtual Number_Command * create_number_command (int num) {
    return new Stack_Number_Command (this->s_, num);
  }

  virtual Add_Command * create_add_command (void) {
    return this->add_;
  }

  // ...

private:
  Stack_Add_Command * add_;
  /
};
```

This approach will improve performance, but you must be mindful about memory management schemes: what objects can and cannot be deleted.

Using Either Abstract Factory

The client uses the infix_to_postfix conversion algorithm as follows:

```
//== traditional version

// get input from STDIN
std::string infix;

// non-flyweight version...
Stack <int> result;
Stack_Expr_Command_Factory factory (result);

Array <Command *> postfix;
infix_to_postfix (infix, factory, postfix);
```

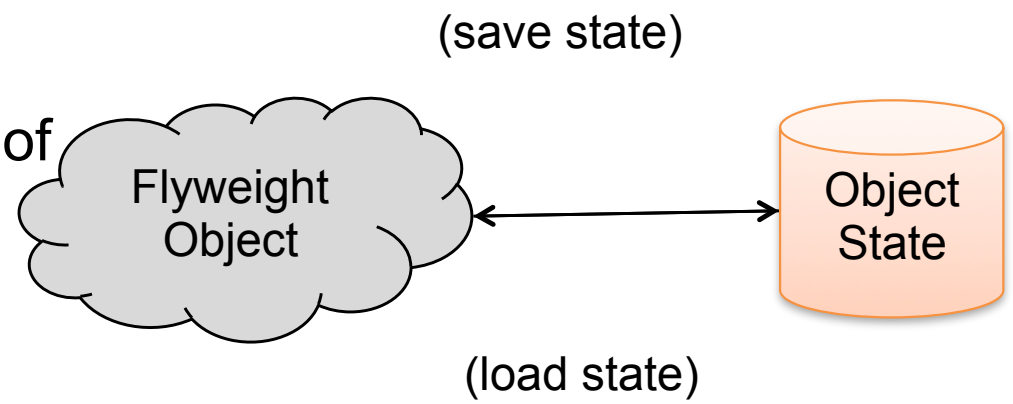
```
//== flyweight version

// get input from STDIN
std::string infix;

Stack <int> result;
Flyweight_Expr_Command_Factory flyweight_factory (result);

Array <Command *> postfix;
infix_to_postfix (infix, flyweight_factory, postfix);
```

The Flyweight Pattern - Consequences

The Flyweight Pattern has the following consequences:

- Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state—especially if it was stored as intrinsic state
- Storage savings are a factor of several functions:
  - Reduction in the total number of instances from sharing
  - Amount of intrinsic state per object
    - The less the better
  - Whether intrinsic state is computed or stored

How much state are we storing for each object instance?



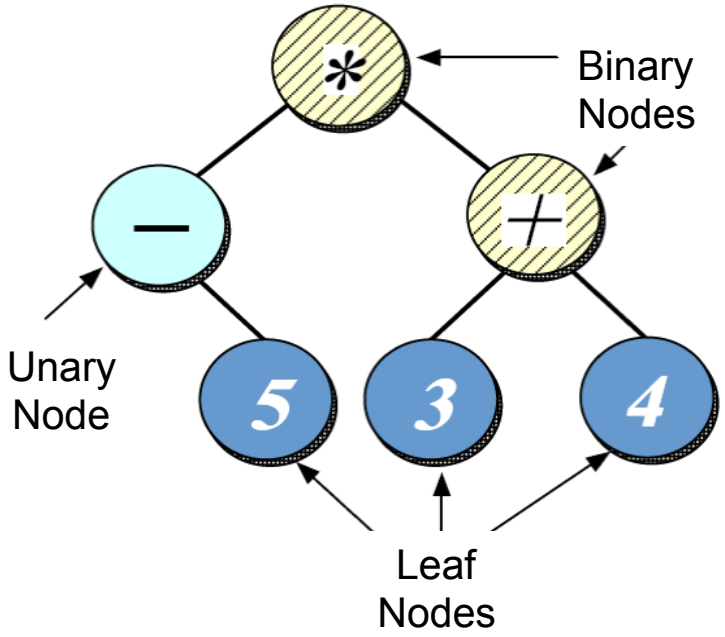
Do I have to compute this state when I reload it into the object?

The Composite Pattern – Motivating Ex.

Our current design for the expression evaluator uses a stack-based implementation the evaluates a postfix expression:

-5 * (3 + 4) ➔ -5 3 4 + *

It is also possible to represent the expression as an expression tree to better show operator association with operands, and use *postorder?* traversal to evaluate it:



The Composite Pattern – Motivation

Have you ever been in the following situation:

- Have a LOT of objects and need to show their hierarchical relations
  - *e.g.*, all the objects in a skyscraper
- Have a LOT of objects and objects composed of many other objects, but really do not care about their implementation
  - *e.g.*, an expression tree vs. a binary node vs. a unary node

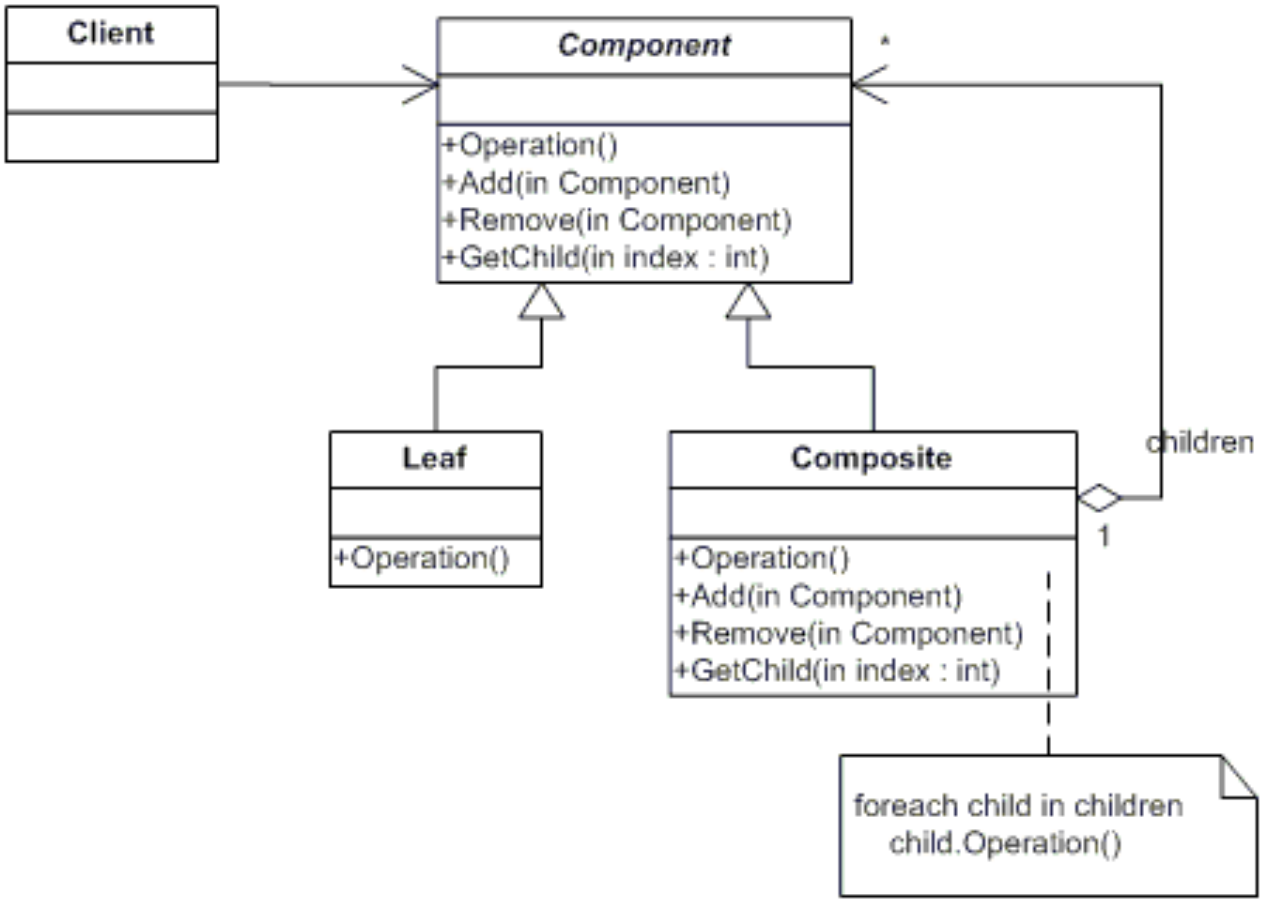
Problem Analogy

A building has floors and floors have rooms. Calculate area of each room, combine area of all rooms on each floor, then combine area of each floor

The Composite Pattern

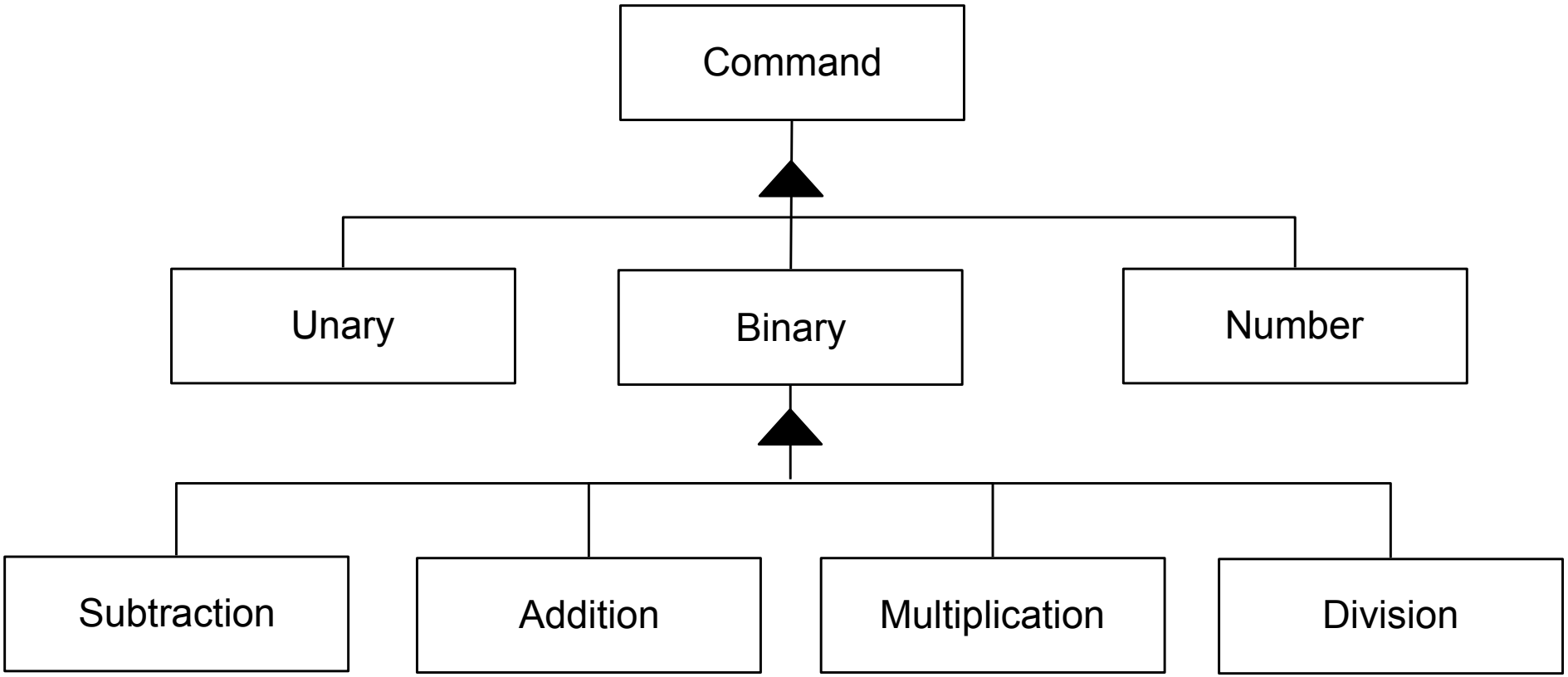
Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly



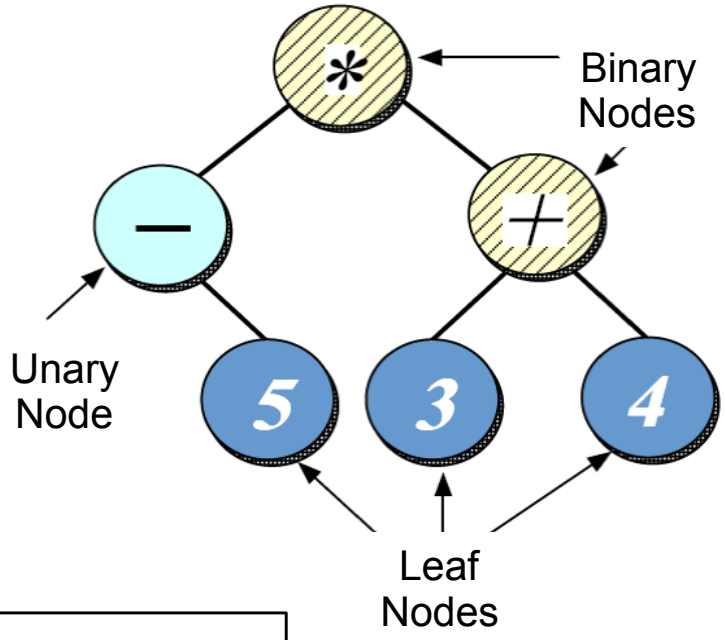
The Composite Pattern - Example

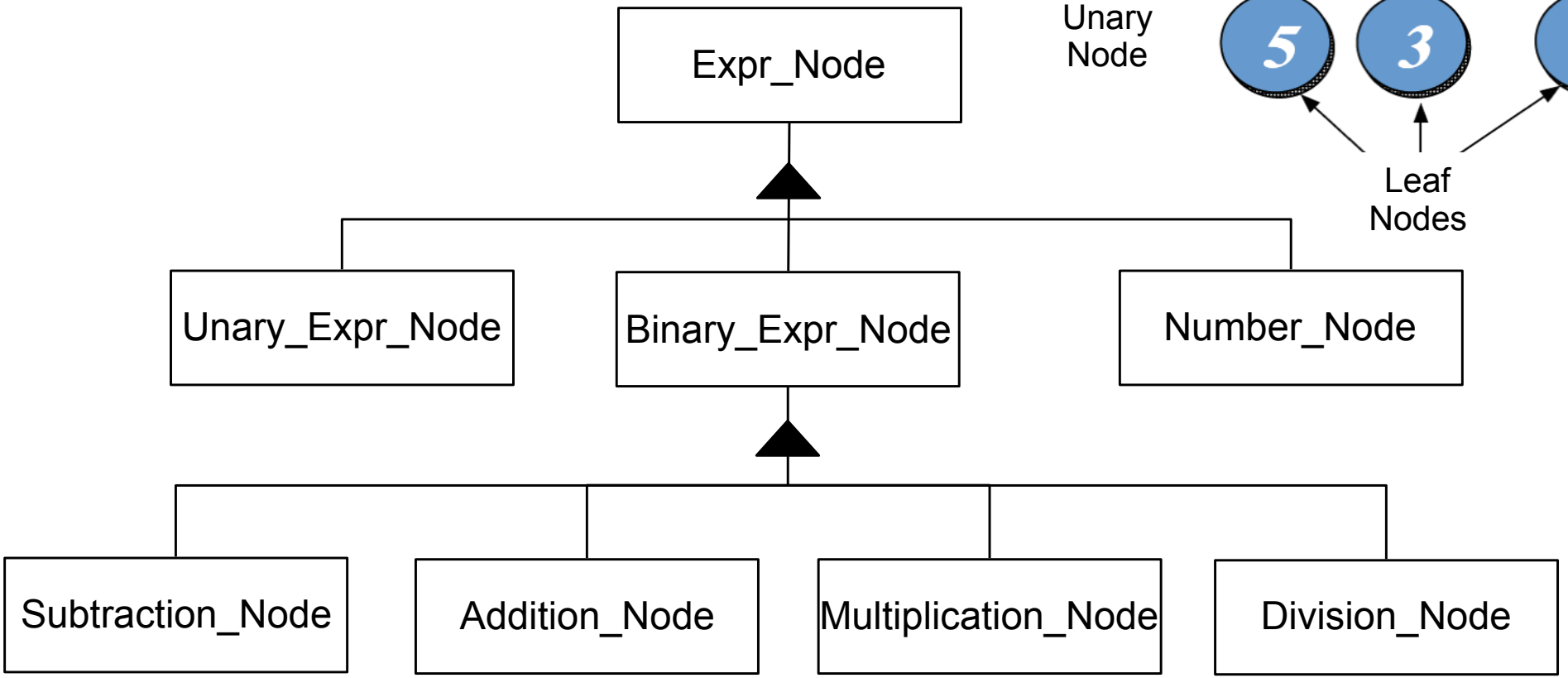
From our previous design, we know that each entity in the expression can be represented as a command object



The Composite Pattern - Example

This same hierarchy can be use to represent each node in the expression tree





The Composite Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual int eval (void) = 0;
};
```

Expr_Node is the base class for all node types in the expression tree

The Composite Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual int eval (void) = 0;
};
```

eval () is "operation" that evaluates each node in the tree (*i.e.*, the composite)

The Composite Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual int eval (void) = 0;
};

class Unary_Expr_Node : public Expr_Node {
public:
  Unary_Expr_Node (void);
  virtual ~Unary_Expr_Node (void);

  virtual int eval (void){
    if (this->child_) return this->child_->eval ();
  }
protected:
  Expr_Node * child_;
};
```

The Unary_Expr_Node only has 1 child

The Composite Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Binary_Expr_Node : public Expr_Node {
public:
  Binary_Expr_Node (void);
  virtual ~Binary_Expr_Node (void);
  // ...
  virtual int eval (void) {
    // use template method to provide common
    // behavior for all binary nodes
  }
protected:
  Expr_Node * right_;
  Expr_Node * left_;
};
```

The Binary_Expr_Node only has 2 children

The Composite Pattern - Example

Let's use the Template Method to provide common functionality to the subclasses that are binary operators

```
class Binary_Expr_Node : public Expr_Node {
public:
  Binary_Expr_Node (void);
  virtual ~Binary_Expr_Node (void);
  // ...
  virtual int eval (void) {
    // use template method to provide common
    // behavior for all binary nodes
  }
protected:
  Expr_Node * right_;
  Expr_Node * left_;
};


class Add_Expr_Node : public Binary_Expr_Node {
public:
  Add_Expr_Node (void);
  virtual ~Add_Expr_Node (void);

  virtual int eval (int num1, int num2);
};
```

Performs the addition evaluation

The Composite Pattern - Example

You can then use the expression tree as follows:

```
// 5 + 4
Expr_Node * n1 = new Number_Node (5);
Expr_Node * n2 = new Number_Node (4);
Expr_Node * expr = new Add_Node (n1, n2);

int result = expr->eval ();
delete expr;
```

eval () calculates the results of the expression tree

The Composite Pattern - Example

You can then use the expression tree as follows:

```
// 5 + 4
Expr_Node * n1 = new Number_Node (5);
Expr_Node * n2 = new Number_Node (4);
Expr_Node * expr = new Add_Node (n1, n2);

int result = expr->eval ();
delete expr;
```

Delete operator will delete all the nodes in the tree

The Composite Pattern - Example

You can then use the expression tree as follows:

```
// 5 + 4
Expr_Node * n1 = new Number_Node (5);
Expr_Node * n2 = new Number_Node (4);
Expr_Node * expr = new Add_Node (n1, n2);

int result = expr->eval ();
delete expr;
```

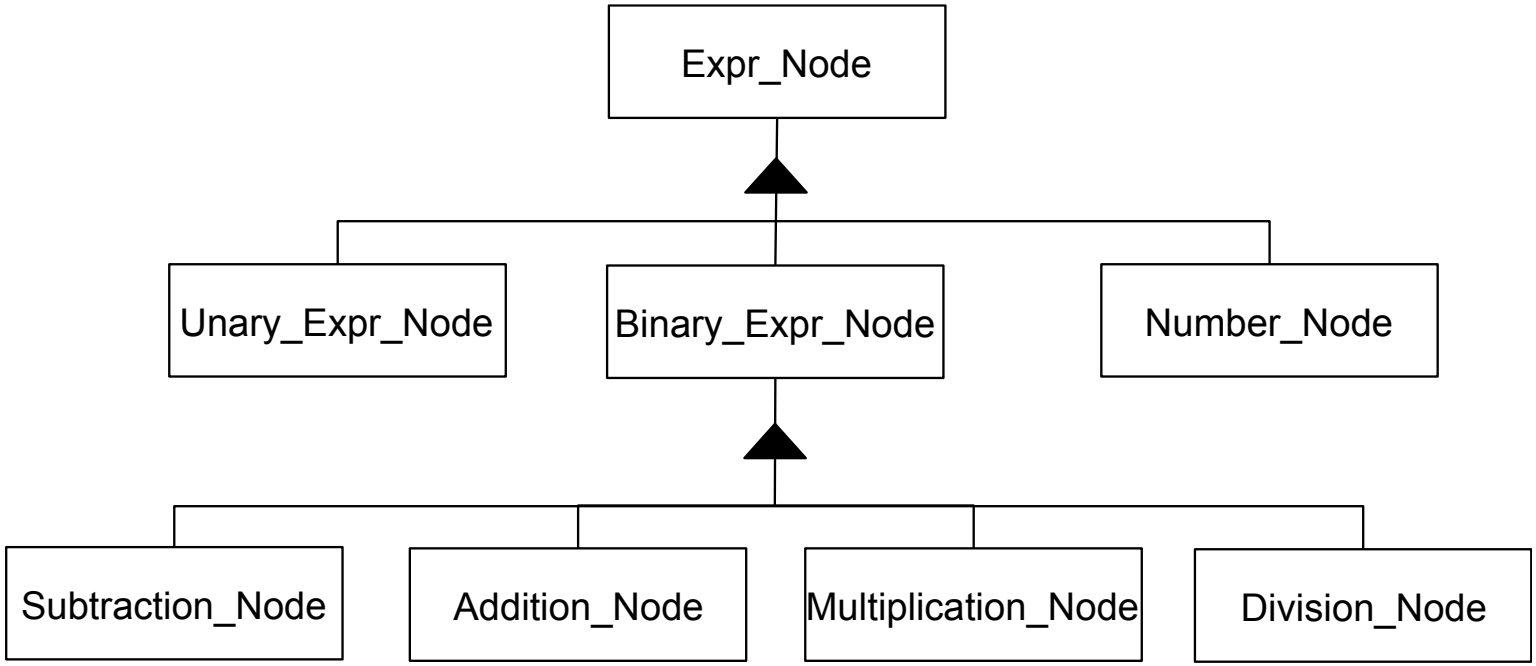
```
// 5 + 4 - 8
Expr_Node * n1 = new Number_Node (5);
Expr_Node * n2 = new Number_Node (4);
Expr_Node * n3 = new Number_Node (8);
Expr_Node * e1 = new Add_Node (n1, n2);
Expr_Node * e2 = new Subtract_Node (e1, n3);

int result = e2->eval ();
delete expr;
```

eval () calculates the results of the expression tree

The Composite Pattern - Consequences

The Composite Pattern has the following consequences:

- Defines class hierarchies consisting of primitive objects
- Makes the client simple
  - Clients can treat composite structures and individual objects uniformly
- Makes it easier to add new kinds of components
- Can make your design overly general
  - *e.g.*, harder to restrict the components of a composite



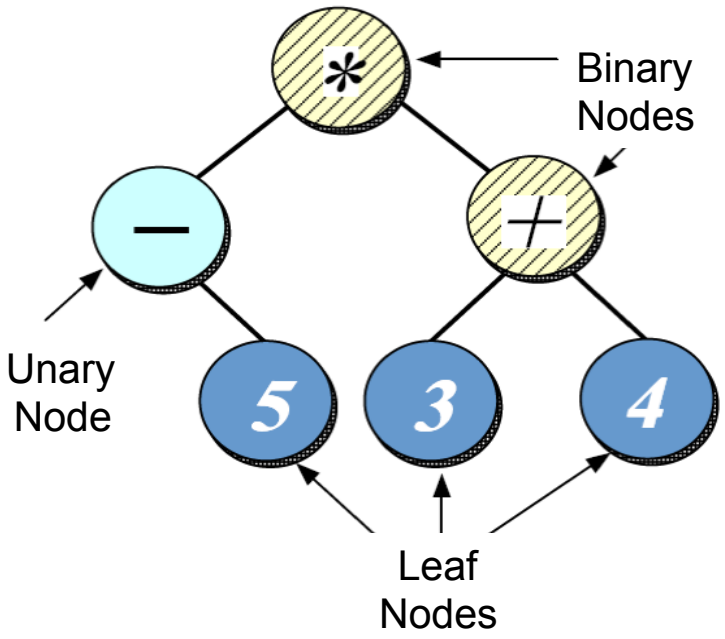
The Visitor Pattern – Motivation

Have you ever been in the following situation:

- Need to traverse a complex structure, but do not want to update the structure to reflect the traversal
  - e.g., traversing a composite
- Have traversal logic that is both complex and must keep track of state during the traversal
  - e.g., coloring a tree

The Visitor Pattern – Motivating Ex.

There are multiple ways to traverse a tree: in-order, post-order, pre-order



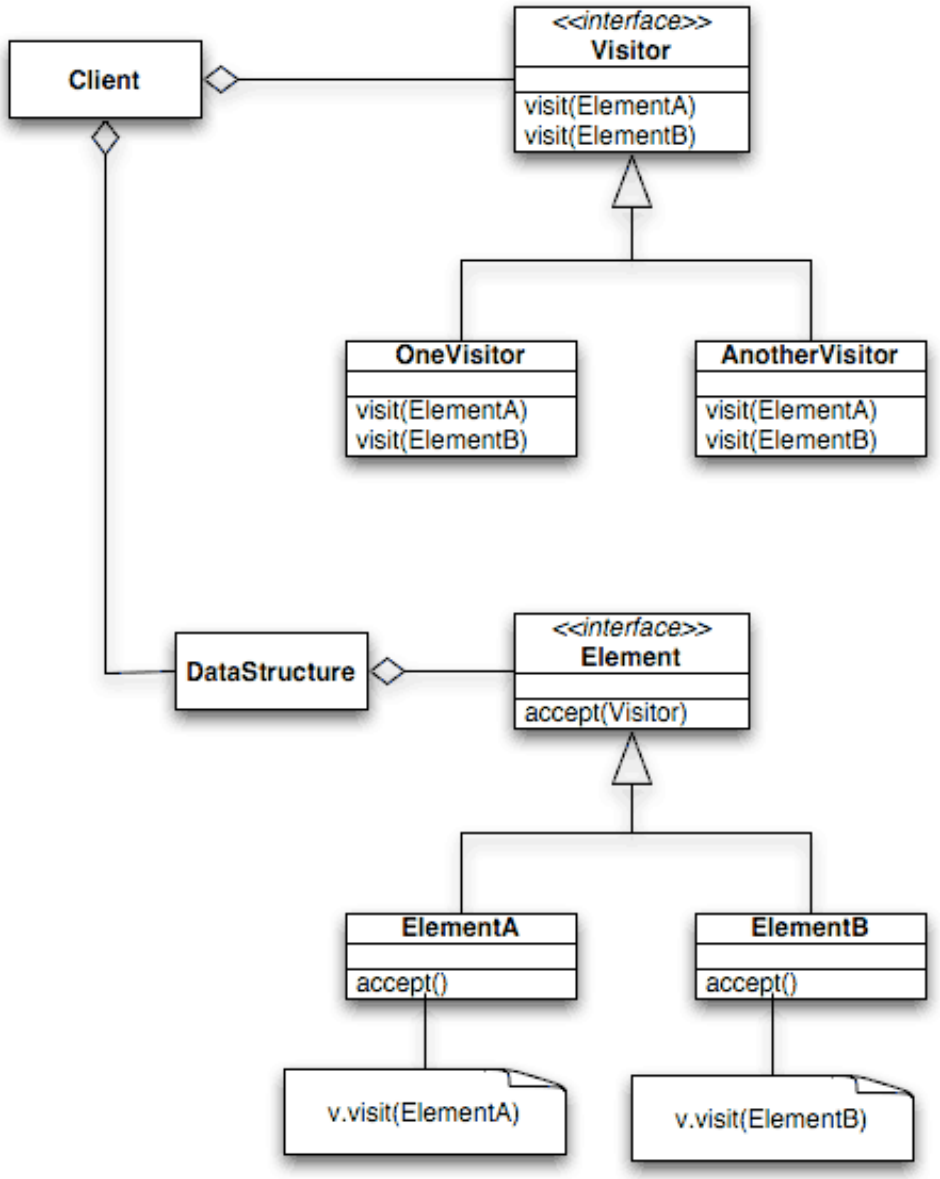

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual int eval (void) = 0;
  virtual void preorder (ostream &) = 0;
};
```

Each method that "traverses" the tree is given a separate method

How can we provide different methods for traversing the tree without having to update the class definition (or structure) each time.

The Visitor Pattern

Intent
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates



The Visitor Pattern – Example

- The current implementation of Expr_Node uses the Composite pattern

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual int eval (void) = 0;
};
```

Visits all nodes in post-order traversal

- The eval () method visits all nodes in the tree using post-order traversal

The Visitor Pattern – Example

- If we want to perform different operations on the expression tree, then we must add the operation to Expr_Node, and implement the operation for each concrete type

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual int eval (void) = 0;
  virtual void print_preorder (ostream &) = 0;
  virtual void print_inorder (ostream &) = 0;
  // ...
};
```

Required implementation of each operation

The Visitor Pattern – Example

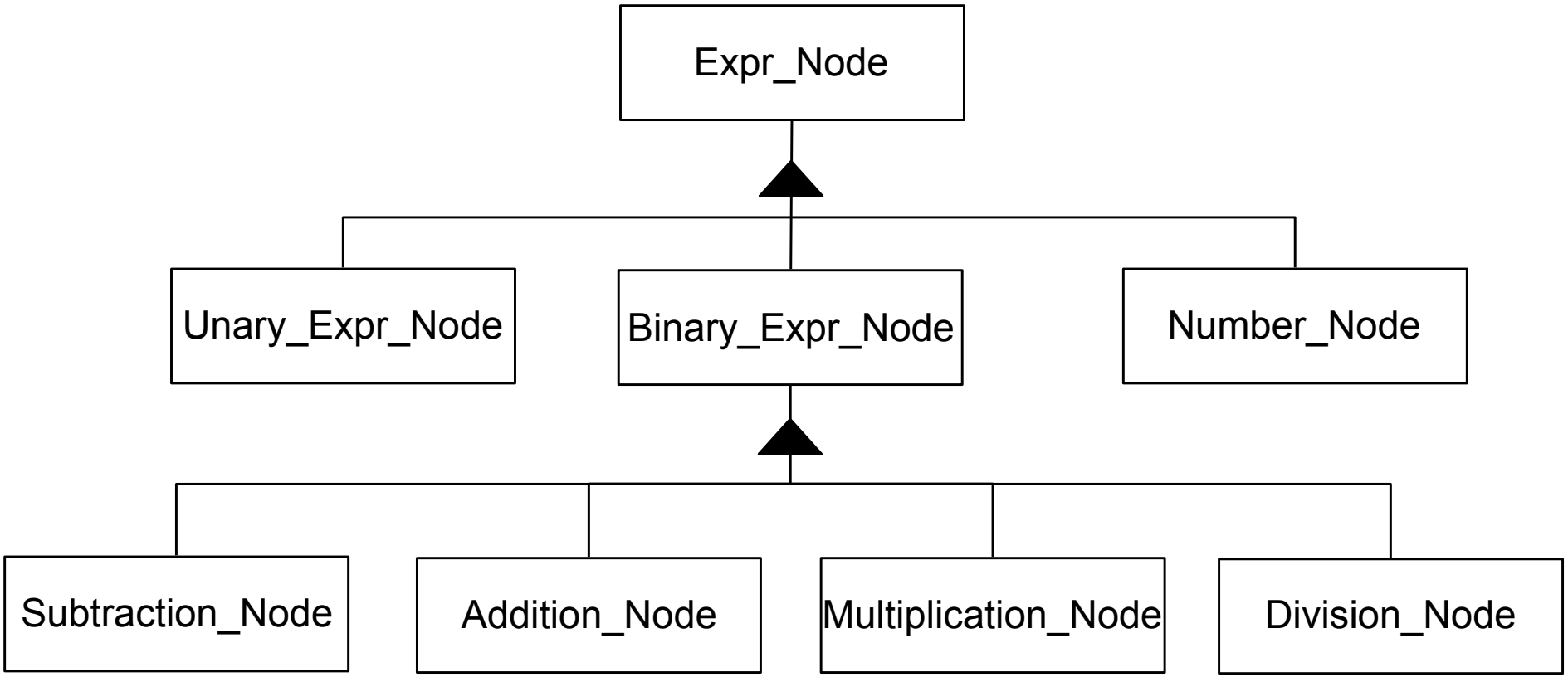
- If we want to perform different operations on the expression tree, then we must add the operation to Expr_Node, and implement the operation for each concrete type

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual int eval (void) = 0;
  virtual void print_preorder (ostream &) = 0;
  virtual void print_inorder (ostream &) = 0;
  // ...
};
```

- Adding new operations causes the class structure to change

The Visitor Pattern – Example

- The Visitor pattern can help prevent the class structure from change for each operation that requires traversing the tree
- The Visitor class defines an interface that visits all (concrete) nodes in the structure



The Visitor Pattern – Example

- The Visitor pattern can help prevent the class structure from change for each operation that requires traversing the tree
- The Visitor class defines an interface that visits all (concrete) nodes in the structure

```
class Expr_Node_Visitor {
public:
  virtual ~Expr_Node_Visitor (void);

  // Methods for visiting concrete nodes
  virtual void Visit_Addition_Node (const Addition_Node & node);
  virtual void Visit_Subtraction_Node (const Subtraction_Node & node);
  virtual void Visit_Number_Node (const Number_Node & node);
  // ...
};
```

Concrete visitors only overload desired methods

The Visitor Pattern – Example

- The Eval_Expr_Tree visitor evaluates the expression tree by visiting elements using post-order traversal logic

```
class Eval_Expr_Tree : public Expr_Node_Visitor {
public:
  Eval_Expr_Tree  (void);
  virtual ~Eval_Expr_Tree (void);

  // Methods for visiting concrete nodes
  virtual void Visit_Addition_Node (const Addition_Node & node) {
    // visit left node, visit right node, then perform addition
  }
  virtual void Visit_Subtraction_Node (const Subtraction_Node & node);
  virtual void Visit_Number_Node (const Number_Node & node);
  // ...

  int result (void) const { return this->result_; }

private:
  int result_;
  // other state for calculating result
};
```

Each Visit_* method contains logic for visiting a concrete node

The Visitor Pattern – Example

- The Eval_Expr_Tree visitor evaluates the expression tree by visiting elements using post-order traversal logic

```
class Eval_Expr_Tree : public Expr_Node_Visitor {
public:
  Eval_Expr_Tree  (void);
  virtual ~Eval_Expr_Tree (void);

  // Methods for visiting concrete nodes
  virtual void Visit_Addition_Node (const Addition_Node & node) {
    // visit left node, visit right node, then perform addition
  }
  virtual void Visit_Subtraction_Node (const Subtraction_Node & node);
  virtual void Visit_Number_Node (const Number_Node & node);
  // ...

  int result (void) const { return this->result_; }

private:
  int result_;
  // other state for calculating result
};
```

Cumulative state for the calculating the result

The Visitor Pattern – Example

- The Eval_Expr_Tree visitor evaluates the expression tree by visiting elements using post-order traversal logic

```
class Eval_Expr_Tree : public Expr_Node_Visitor {
public:
  Eval_Expr_Tree  (void);
  virtual ~Eval_Expr_Tree (void);

  // Methods for visiting concrete nodes
  virtual void Visit_Addition_Node (const Addition_Node & node) {
    // visit left node, visit right node, then perform addition
  }
  virtual void Visit_Subtraction_Node (const Subtraction_Node & node);
  virtual void Visit_Number_Node (const Number_Node & node);
  // ...

  int result (void) const { return this->result_; }

private:
  int result_;
  // other state for calculating result
};
```

The result () method returns the result of the evaluation

The Visitor Pattern – Example

- The Print_Inorder_Expression visitor prints the expression as an infix expression to the specified output

```
class Print_Inorder_Expression : public Expr_Node_Visitor {
public:
  Print_Inorder_Expression (std::ostream & out)
   : out_ (out) { }
  virtual ~Print_Inorder_Expression (void);

  // Methods for visiting concrete nodes
  virtual void Visit_Addition_Node (const Addition_Node & node);
  virtual void Visit_Subtraction_Node (const Subtraction_Node & node);
  virtual void Visit_Number_Node (const Number_Node & node);
  // ...

private:
  // output stream
  std::ostream out_;
};
```

Print_Inorder_Expression subclasses Expr_Node_Visitor, but contains it own behavior/state

The Visitor Pattern – Example

- The Print_Inorder_Expression visitor prints the expression as an infix expression to the specified output

```
class Print_Inorder_Expression : public Expr_Node_Visitor {
public:
  Print_Inorder_Expression (std::ostream & out)
   : out_ (out) { }
  virtual ~Print_Inorder_Expression (void);

  // Methods for visiting concrete nodes
  virtual void Visit_Addition_Node (const Addition_Node & node);
  virtual void Visit_Subtraction_Node (const Subtraction_Node & node);
  virtual void Visit_Number_Node (const Number_Node & node);
  // ...

private:
  // output stream
  std::ostream out_;
};
```

- The same approach can be used for other concrete visitors

The Visitor Pattern – Example

- Adding visitor support to the classes requires adding an accept () method

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual void accept (Expr_Node_Visitor & v) = 0;
};
```

Accept method take a visitor as its only input parameter

The Visitor Pattern – Example

- Adding visitor support to the classes requires adding an accept () method

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual void accept (Expr_Node_Visitor & v) = 0;
};
```

- Concrete classes implement the accept () method and pass control to the specified visitor

```
class Addition_Node : Binary_Expr_Node {
public:
  // ...

  virtual void accept (Expr_Node_Visitor & v) {
    v.Visit_Addition_Node (*this);
  }
};
```

Accept method passes control to concrete visitor

The Visitor Pattern – Example

- To use the visitor, just create a concrete visitor and pass it to the accept method

```
// ...
Expr_Node * expr_tree = /* get tree from somewhere */

Eval_Expr_Tree eval;

// evaluate the expression tree
expr_tree->accept (eval);
int result = eval.result ();

// print the tree in infix format
Print_Inorder_Expression printer (std::cout);
expr_tree->accept (printer);
```

Get pointer to the expression tree

The Visitor Pattern – Example

- To use the visitor, just create a concrete visitor and pass it to the accept method

```
// ...
Expr_Node * expr_tree = /* get tree from somewhere */

Eval_Expr_Tree eval;

// evaluate the expression tree
expr_tree->accept (eval);
int result = eval.result ();

// print the tree in infix format
Print_Inorder_Expression printer (std::cout);
expr_tree->accept (printer);
```

Get result from visiting all nodes

The Visitor Pattern – Example

- To use the visitor, just create a concrete visitor and pass it to the accept method

```
// ...
Expr_Node * expr_tree = /* get tree from somewhere */

Eval_Expr_Tree eval;

// evaluate the expression tree
expr_tree->accept (eval);
int result = eval.result ();

// print the tree in infix format
Print_Inorder_Expression printer (std::cout);
expr_tree->accept (printer);
```

Pass visitor to accept () method

The Visitor Pattern - Consequences

The Visitor Pattern has the following consequences:

- Visitors make it easy to add operations that depend on components of complex objects

```
class Expr_Node {
public:
  Expr_Node (void);
  virtual ~Expr_Node (void);

  // Used to traverse the tree
  virtual void
  accept (Expr_Node_Visitor &) = 0;
};

// Operations are defined as concrete
// visitor of Expr_Node_Visitor
class Expr_Node_Visitor;
```

The Visitor Pattern - Consequences

The Visitor Pattern has the following consequences:

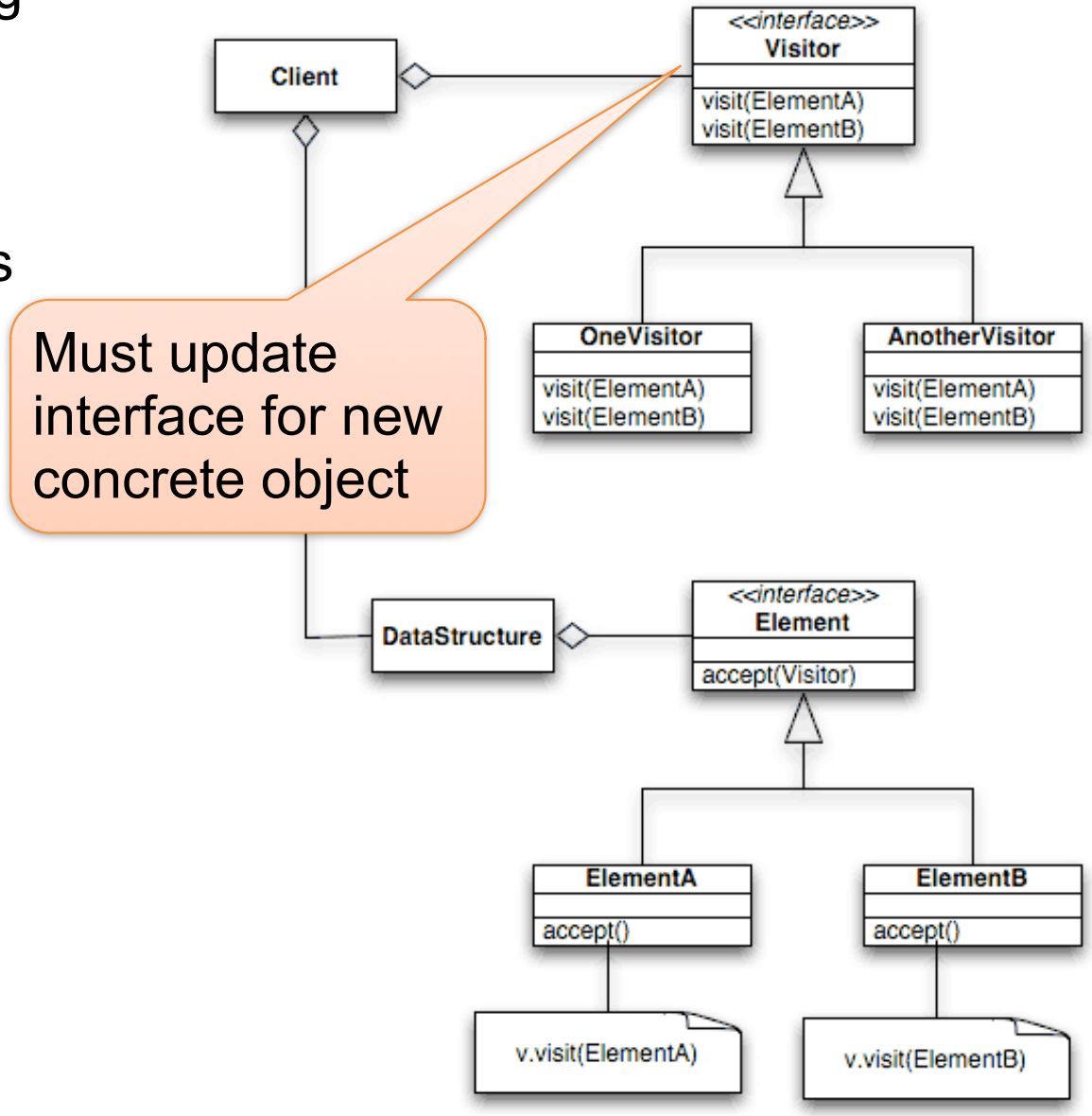
- Visitors make it easy to add operations that depend on components of complex objects
- A visitor gathers related operations and separates unrelated ones
  - i.e., related behavior is localized to the visitor
- Accumulating state

```
class Print_Inorder_Expression :
  public Expr_Node_Visitor {
public:

  // Methods for visiting concrete nodes

private:
  // State required for visitation
  std::ostream out_;
};
```

The Visitor Pattern - Consequences

The Visitor Pattern has the following consequences:

- Visitors make it easy to add operations that depend on components of complex objects
- A visitor gathers related operations and separates unrelated ones
  - i.e., related behavior is localized to the visitor
- Accumulating state
- Adding new concrete element classes is hard



The Visitor Pattern - Consequences

The Visitor Pattern has the following consequences:

- Visitors make it easy to add operations that depend on components of complex objects
- A visitor gathers related operations and separates unrelated ones
  - i.e., related behavior is localized to the visitor
- Accumulating state
- Adding new concrete element classes is hard
- Visiting across class hierarchies

```
class A;

class B;

class Visitor {
public:
  virtual void Visit_A (A & a);
  virtual void Visit_B (B & b);
};
```

A and B are not in same hierarchy, but visitor still functions as expected

The Visitor Pattern - Consequences

The Visitor Pattern has the following consequences:

- Visitors make it easy to add operations that depend on components of complex objects
- A visitor gathers related operations and separates unrelated ones
  - i.e., related behavior is localized to the visitor
- Accumulating state
- Adding new concrete element classes is hard
- Visiting across class hierarchies
- Can compromise encapsulation

May have to expose the internal state of A and B to visit its children.

```
class A;

class B;

class Visitor {
public:
  virtual void Visit_A (A & a);
  virtual void Visit_B (B & b);
};
```

The Visitor Pattern - Implementation

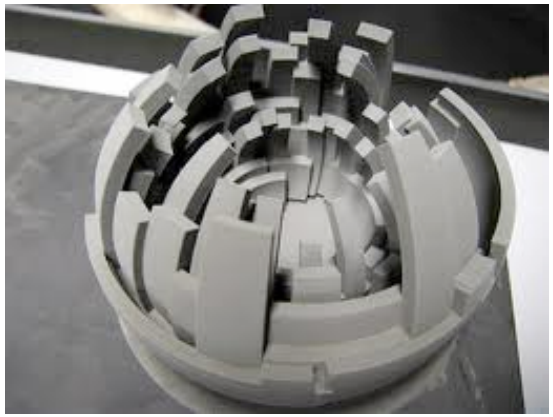
The following are implementation issues that arise when implementing the Visitor pattern:

- Double dispatch

```
class A : public Visitable {
  virtual void Accept (V & v) {
    v.Visit_A (*this);
  }
}

class Visitor {
public:
  virtual void Visit_A (A & a);
};
```

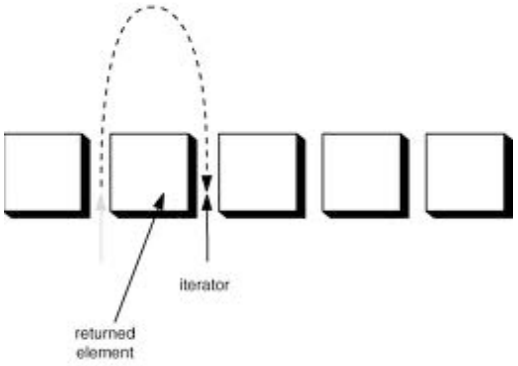
Double dispatch

- Who is responsible for traversing the object structure?



The Object Structure

The Visitor



Separate Iterator

The Builder Pattern – Motivation

Have you ever been in the following situation:

- Need to build a complex object, but want shield the client from the complexity of building the object
    - e.g., converting an infix expression to a binary tree
- Have many ways of building a the same abstraction, but with different internal representations
    - e.g., postfix vs. binary tree representation of a infix expression

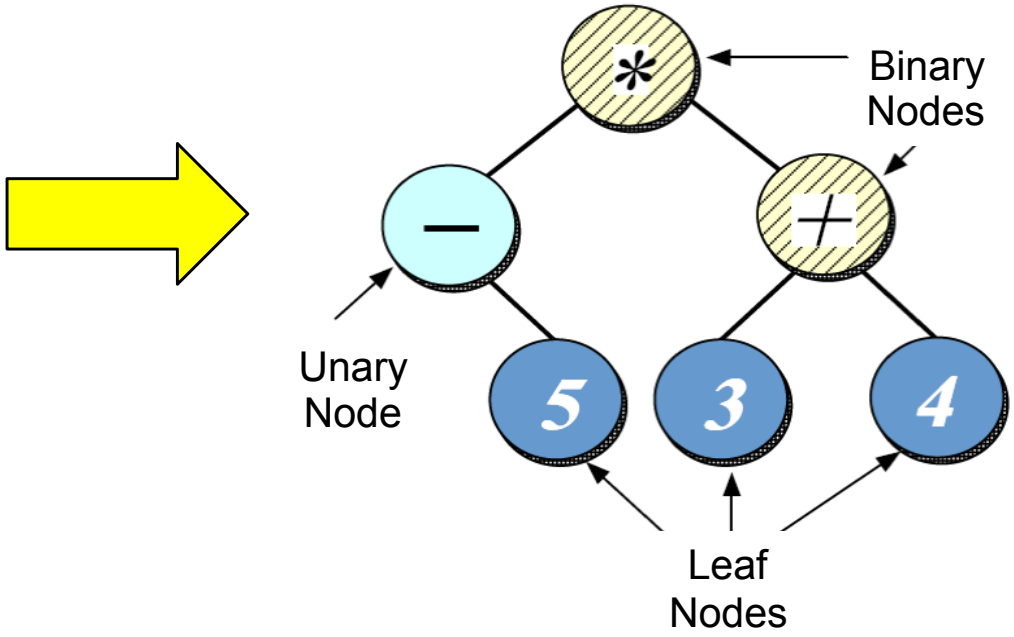
Problem Analogy

Each “builder” has their own method for constructing a home. The homeowner, however, does not care how the home is constructed. The homeowner just wants their home built as promised.

The Builder Pattern – Motivating Ex.

Converting the index expression to a binary tree to is a non-trivial task.

-5 * (3 + 4)

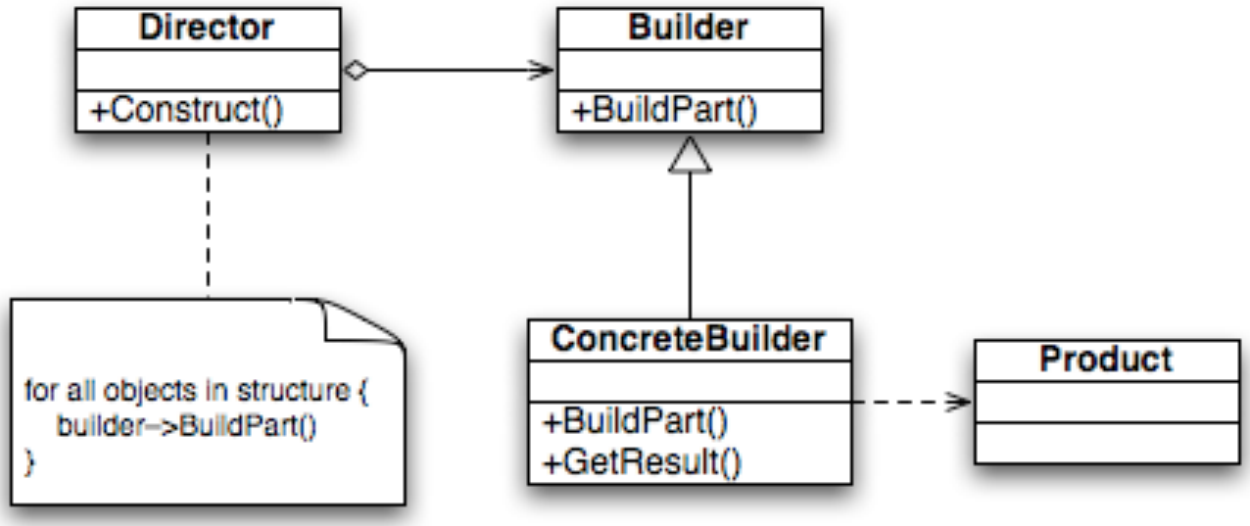


How can we shield the client from the process of creating a binary tree from an infix expression, and support other representations of the infix expression, such as a postfix representation?

The Builder Pattern

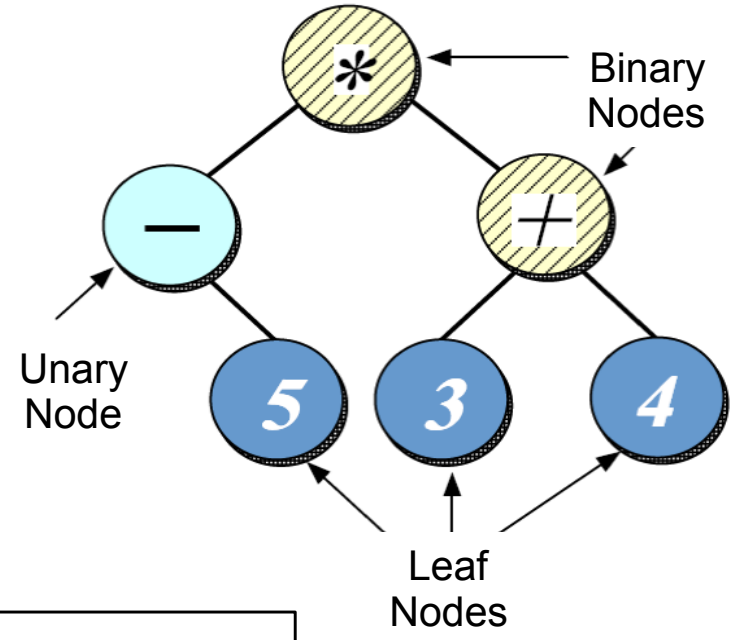
Intent

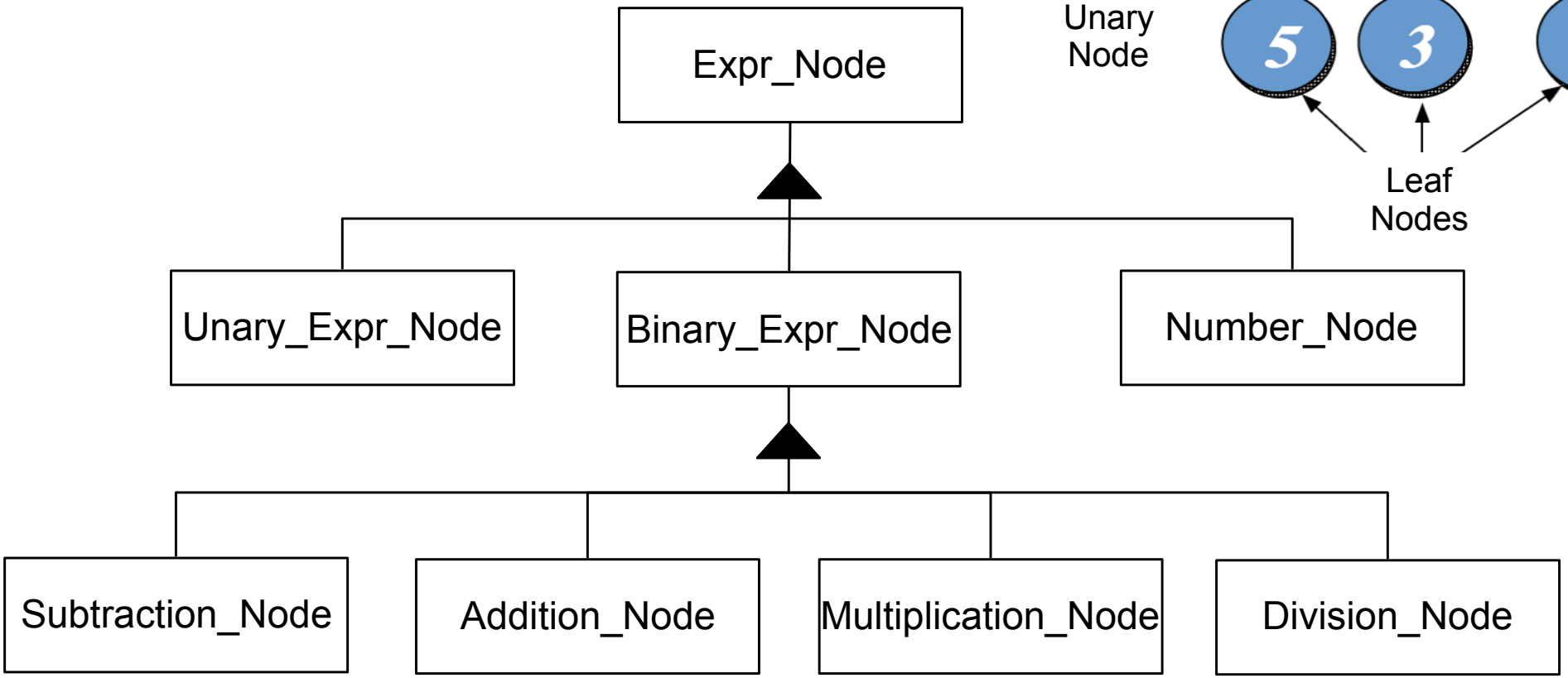
Separate the construction of a complex object from its representation so that the same construction process can create different representations



The Builder Pattern - Example

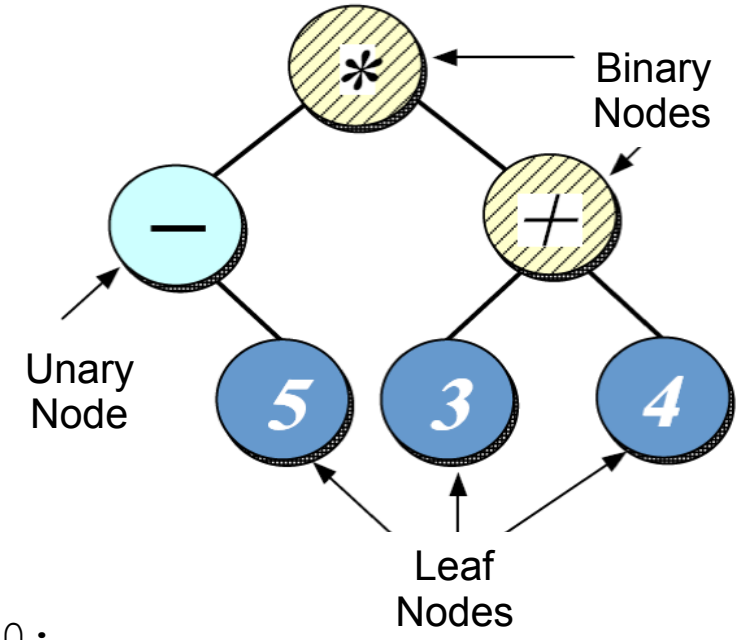
The current representation of the each node in the expression tree is a follows:





The Builder Pattern - Example

- Given the structural design of the expression tree, we know that the builder will have the following interface




```
class Expr_Builder {
public:
  // ...
  virtual void start_expression (void);
  virtual void build_number (int n) = 0;
  virtual void build_add_operator (void) = 0;
  virtual void build_subtract_operator (void) = 0;
  // ...
  virtual void build_open_parenthesis (void) = 0;
  virtual void build_close_parenthesis (void) = 0;

  // get the current expression
  virtual Math_Expr * get_expression (void) = 0;
};
```

- Each method is responsible for building some part of the expression tree without requiring the client (or director) to understand the internal representation of the final expression

The Builder Pattern - Example

- To use the builder, we integrate it with the parser that knows how to parse the expression, which is the “director” for this pattern

```
bool Calculator::parse_expr (const string & infix)
{
  std::istringstream input (infix); // create a input stream parser
  std::string token;                // current token in string/stream

  b.start_expression ();            // start a new expression

  while (!input.eof ()) {
     input >> token;
     if (token == "+") this->builder_.build_add_operator ();
     else if (token == "-") this->builder_.build_subtract_operator ();
     else if (token == "*") this->builder_.build_multiply_operator ();
     // ...
  }

  return true;
};
```

- Again, the client does not know the exact representation of final expression

The Builder Pattern - Example

- To use the builder, we integrate it with the parser that knows how to parse the expression, which is the “director” for this pattern

```
bool Calculator::parse_expr (const string & infix)
{
  std::istringstream input (infix); // create a input stream parser
  std::string token;                // current token in string/stream

  b.start_expression ();            // start a new expression

  while (!input.eo    ()) {
     input >> toke
     if (token       der_.build_add_operator ();
     else if (t      >builder_.build_subtract_operator ();
     else if (t      >builder_.build_multiply_operator ();
     // ...
  }

  return true;
};
```

Build are initial expression tree

- Again, the client does not know the exact representation of final expression

The Builder Pattern - Example

- To use the builder, we integrate it with the parser that knows how to parse the expression, which is the “director” for this pattern

```
bool Calculator::parse_expr (const string & infix)
{
  std::istringstream input (infix); // create a input stream parser
  std::string token;                                  en in string/stream

  b.start_expression ();                               expression

  while (!input.eof ()) {
     input >> token;
     if (token == "+") this->builder_.build_add_operator ();
     else if (token == "-") this->builder_.build_subtract_operator ();
     else if (token == "*") this->builder_.build_multiply_operator ();
     // ...
  }

  return true;
};
```

Parser directs builder how to build expression

- Again, the client does not know the exact representation of final expression

The Builder Pattern - Example

- When creating the builder for the binary tree representation of the expression, we subclass the builder for our needs

```
class Expr_Tree_Builder : public Expr_Builder {
  Expr_Tree_Builder (void);
  virtual ~Expr_Builder_Tree (void);

  virtual void start_expression (void) {
    // ...
    this->tree_ = new Expr_Tree ();
  }

  virtual void build_number (int n);
  virtual void build_add_operator (void);
  virtual void build_subtract_operator (void);
  // ...

  Expr_Tree * get_expression (void) { return this->tree_; }

private:
  // current state of expression tree
  Expr_Tree * tree_;
  // other variables to coordinate build process
};
```

The Builder Pattern - Example

- When creating the builder for the binary tree representation of the expression, we subclass the builder for our needs

```
class Expr_Tree_Builder : public Expr_Builder {
  Expr_Tree_Builder (void);
  virtual ~Expr_Builder_Tree (void);

  virtual void start_expression (void) {
    // ...
    this->tree_ =
  }

  virtual void build_number (int n);
  virtual void build_add_operator (void);
  virtual void build_subtract_operator (void);
  // ...

  Expr_Tree * get_expression (void) { return this->tree_; }

private:
  // current state of expression tree
  Expr_Tree * tree_;
  // other variables to coordinate build process
};
```

Expr_Tree_Builder is a concrete version of the Expr_Builder

The Builder Pattern - Example

- When creating the builder for the binary tree representation of the expression, we subclass the builder for our needs

```
class Expr_Tree_Builder : public Expr_Builder {
  Expr_Tree_Builder (void);
  virtual ~Expr_Builder_Tree (void);

  virtual void start_expression (void) {
    // ...
    this->tree_ = new Expr_Tree ();
  }

  virtual void build    ber (int n);
  virtual void b
  virtual void b                                    oid);
  // ...

  Expr_Tree * get_expression (void) { return this->tree_; }

private:
  // current state of expression tree
  Expr_Tree * tree_;
  // other variables to coordinate build process
};
```

Create a new Expr_Tree

The Builder Pattern - Example

- When creating the builder for the binary tree representation of the expression, we subclass the builder for our needs

```
class Expr_Tree_Builder : public Expr_Builder {
  Expr_Tree_Builder (void);
  virtual ~Expr_Builder_Tree (void);

  virtual void start_expression (void) {
    // ...
    this->tree_ = new Expr_Tree ();
  }

  virtual void build_number (int n);
  virtual void build_add_operator (void);
  virtual void build_subtract_operator (void);
  // ...

  Expr_Tree * get_                    return this->tree_; }

private:
  // current state of expression tree
  Expr_Tree * tree_;
  // other variables to coordinate build process
};
```

Build nodes on the expression tree

The Builder Pattern - Example

- When creating the builder for the binary tree representation of the expression, we subclass the builder for our needs

```
class Expr_Tree_Builder : public Expr_Builder {
  Expr_Tree_Builder (void);
  virtual ~Expr_Builder_Tree (void);

  virtual void start_expression (void) {
    // ...
    this->tree_ = new Expr_Tree ();
  }

  virtual void build_numbe
  virtual void build_add_
  virtual void build_subtrac    erator (void);
  // ...

  Expr_Tree * get_expression (void) { return this->tree_; }

private:
  // current state of expression tree
  Expr_Tree * tree_;
  // other variables to coordinate build process
};
```

Get the current expression tree

The Builder Pattern - Example

- When creating the builder for the binary tree representation of the expression, we subclass the builder for our needs

```
class Expr_Tree_Builder : public Expr_Builder {
  Expr_Tree_Builder (void);
  virtual ~Expr_Builder_Tree (void);

  virtual void start_expression (void) {
    // ...
    this->tree_ = new Expr_Tree ();
  }

  virtual void build_number (int n);
  virtual void                (void);
  virtual void                rator (void);
  // ...

  Expr_Tree * get    ression (void) { return this->tree_; }

private:
  // current state of expression tree
  Expr_Tree * tree_;
  // other variables to coordinate build process
};
```

State variables for coordinating tree construction

The Builder Pattern - Example

- Finally, you use the Expr_Tree_Builder in client code as follows:

```
Calculator::Calculator (Expression_Builder & builder)
: builder_ (builder) { }

int Calculator::evaluate (const std::string & infix)
{
 if (!this->parse_expr (infix))
    return false;

  std::unique_ptr <Math_Expr> expr (this->builder_—>get_expression ());

  if (nullptr == expr.get ())
    return false;

  // evaluate the expression
  return expr->eval ();
}
```

The Builder Pattern - Example

- Finally, you use the Expr_Tree_Builder in client code as follows:

```
Calculator::Calculator (Expression_Builder & builder)
: builder_ (builder) { }

int Calculator::evaluate (const std::string & infix)
{
 if (!this->parse_expr (infix))
    return false;

  std::unique_ptr <Math                    ilder_->get_expression ());

  if (nullptr == e
    return false;

  // evaluate the expression
  return expr->eval ();
}
```

The expression is parsed and the builder constructs its representation

The Builder Pattern - Example

- Finally, you use the Expr_Tree_Builder in client code as follows:

```
Calculator::Calculator (Expression_Builder & builder)
: builder_ (builder) { }

int Calculator::evaluate (const std::string & infix)
{
 if (!this->parse_expr (infix))
    return false;

  std::unique_ptr <Math_Expr> expr (this->builder_—>get_expression ());

  if (nullptr == expr.get ())
    return false;

  // evaluate the expression
  return expr->eval ();
}
```

The client retrieves expression from the builder

The Builder Pattern - Example

- Finally, you use the Expr_Tree_Builder in client code as follows:

```
Calculator::Calculator (Expression_Builder & builder)
: builder_ (builder) { }

int Calculator::evaluate (const std::string & infix)
{
 if (!this->parse_expr (infix))
    return false;

  std::unique_ptr <Math_Expr> expr (this->builder_—>get_expression ());

  if (nullptr == expr.get ())
    return false;

  // evaluate the expression
  return expr->eval ();
}
```

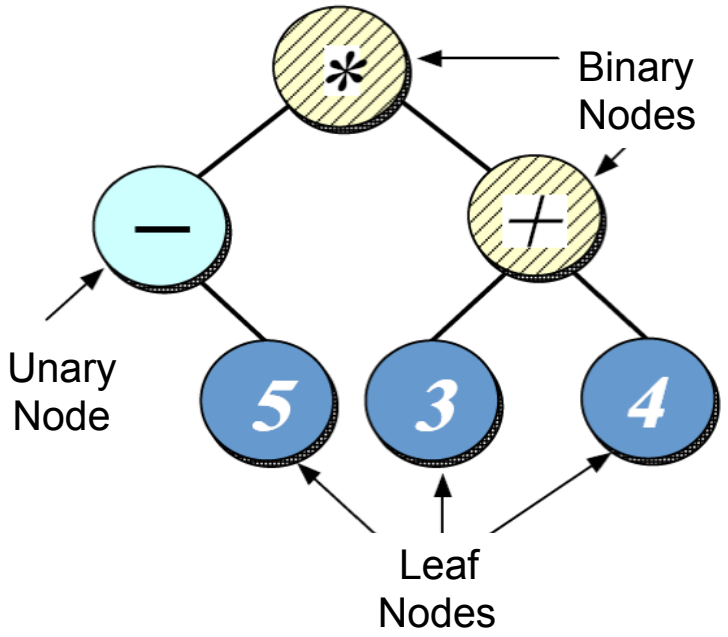
The client then evaluates the expression, and gets the result

The Builder Pattern - Consequences

The Builder Pattern has the following consequences:

- Let's you vary a product's internal representation

Tree Representation



Postfix Representation

-5	3	4	+	*

The Builder Pattern - Consequences

The Builder Pattern has the following consequences:

- Let's you vary a product's internal representation
- It isolates code for construction and representation
  - *i.e.*, separation of concerns, how you build the object is not tightly coupled with the object's representation

```
b.build_expression ();

while (!input.eof ()) {
   input >> token;
   if (token == "+")
     b.build_add_operand ();
   else if (token == "-")
     b.build_subtract_command ();
   else if (token == "*")
     b.build_multiply_command ();
   // ...
}
```

The client only instructs how to build, not how to represent expression

The Builder Pattern - Consequences

The Builder Pattern has the following consequences:

- Let's you vary a product's internal representation
- It isolates code for construction and representation
  - *i.e.*, separation of concerns, how you build the object is not tightly coupled with the object's representation
- Gives you finer control over the construction process
  - it receives instructions from another object (*e.g.*, a parser) instead of creating the object by itself

```
b.build_expression ();

while (!input.eof ()) {
   input >> token;
   if (token == "+")
     b.build_add_operand ();
   else if (token == "-")
     b.build_subtract_command ();
   else if (token == "*")
     b.build_multiply_command ();
   // ...
}
```