Implementing STL Iterator Semantics

Goals

- Ensure the proper semantics of post-increment operations for STL-based **`ET_Iterator`** objects, i.e.:
  - **`Expression_Tree value = iter++`** vs. **`Expression_Tree value = ++iter`**

Implementing STL Iterator Semantics

Goals

- Ensure the proper semantics of post-increment operations for STL-based `ET_Iterator` objects
  - `Expression_Tree value = iter++` vs. `Expression_Tree value = ++iter`

Constraints / forces

- STL pre-increment operations are easy to implement since they simply increment the value & return *this, e.g.,

```
iterator &operator++() { ++...; return *this; }
```

Implementing STL Iterator Semantics

Goals

- Ensure the proper semantics of post-increment operations for STL-based `ET_Iterator` objects
  - `Expression_Tree value = iter++` vs. `Expression_Tree value = ++iter`

Constraints / forces

- STL pre-increment operations are easy to implement since they simply increment the value & return *this, e.g.,

```
iterator &operator++() { ++...; return *this; }
```

- STL post-increment operations are more complicated since they return a copy of the existing iterator *before* incrementing its value, e.g.,

```
iterator operator++(int)
{ iterator temp = copy_*this; ++...; return temp; }
```

Implementing STL Iterator Semantics

Goals

- Ensure the proper semantics of post-increment operations for STL-based `ET_Iterator` objects
  - `Expression_Tree value = iter++` vs. `Expression_Tree value = ++iter`

Constraints / forces

- STL pre-increment operations are easy to implement since they simply increment the value & return *this, e.g.,

```
iterator &operator++() { ++...; return *this; }
```

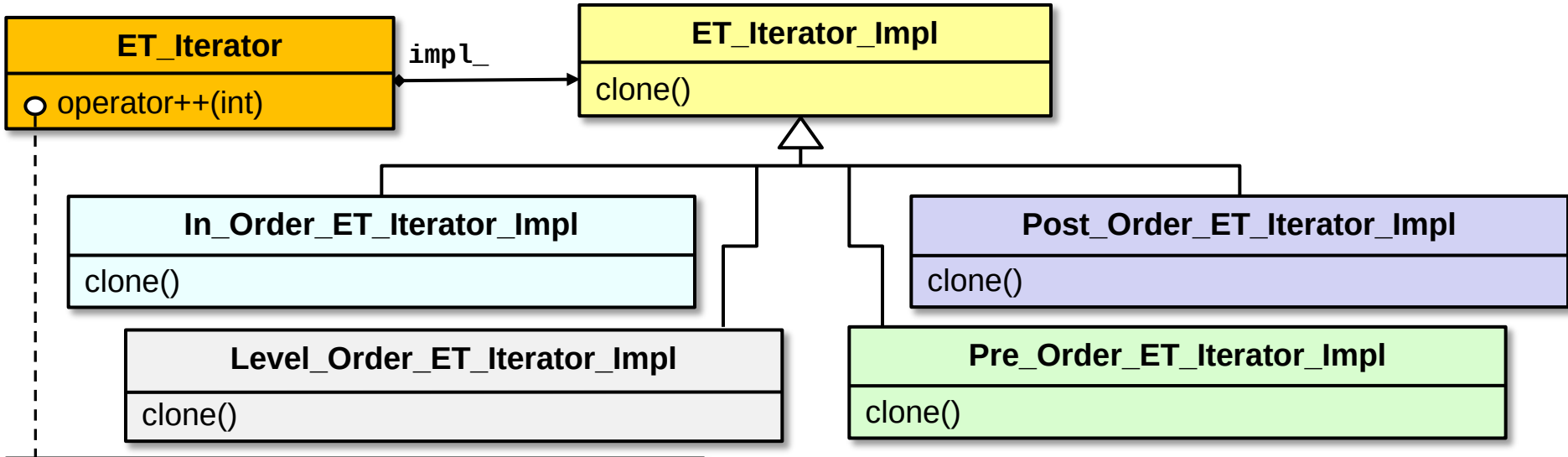
- STL post-increment operations are more complicated since they return a copy of the existing iterator *before* incrementing its value, e.g.,

```
iterator operator++(int)
{ iterator temp = copy_*this; ++...; return temp; }
```

- Since our **`ET_Iterator`** objects use the *Bridge* pattern it is tricky to implement the "***`copy_*this`***" step above in a generic way

As a general rule, in STL it's better to say ++iter than iter++

Solution: Clone a New Instance Via a Prototype




```
iterator
ET_Iterator::operator++(int)
{
  iterator temp(impl_->clone());
  ++(*impl_);
  return temp;
}
```

The *Bridge* pattern abstraction class needn't have direct knowledge of implementor subclass details

Bridge pattern encapsulates variability & simplifies memory management

ET_Iterator_Impl Class Interface

- Subclasses of this base class define various iterations algorithms that can traverse an expression tree

Interface

```
                    ET_Iterator_Impl
                              (const Expression_Tree &tree)
            virtual
  Component_Node * operator *()=0
      virtual void operator++()=0
      virtual bool operator==(const ET_Iterator_Impl &)=0
      virtual bool operator!= (const ET_Iterator_Impl &)=0
            virtual
ET_Iterator_Impl * clone()=0
```

Subclass performs a deep copy

- **Commonality**: Provides a common interface for expression tree iterator implementations
- **Variability**: Each subclass implements the **`clone()`** method to return a deep copy of itself for use by **`ET_Iterator::operator++(int)`**

Prototype GoF Object Creational

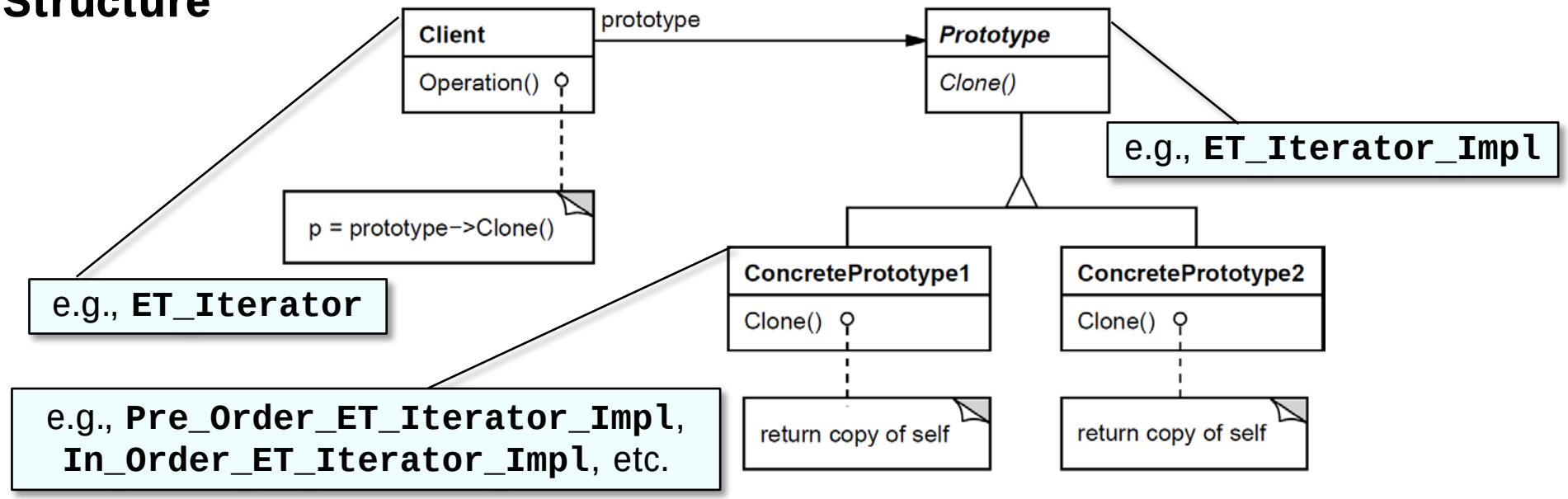
Intent

- Specify the kinds of objects to create using a prototypical instance & create new objects by copying this prototype

Applicability

- When the classes to instantiate are specified at run-time
- There's a need to avoid the creation of a factory hierarchy
- It is more convenient to copy an existing instance than to create a new one

Structure



Prototype GoF Object Creational

Prototype example in C++

- Relationship between iterator interface (*Bridge*) & implementor hierarchy (*Prototype*)

```
iterator
ET_Iterator::operator++(int)
{
  iterator temp(impl_->clone());
  ++(*impl_);
  return temp;
}
```

The *Bridge* pattern abstraction class calls clone() on the implementor subclass to get a deep copy without breaking encapsulation

Prototype GoF Object Creational

Prototype example in C++

- Relationship between iterator interface (*Bridge*) & implementation (*Prototype*)

```
iterator
ET_Iterator::operator++(int)
{
  iterator temp(impl_->clone());
  ++(*impl_);
  return temp;
}
```

This method encapsulates the details of making a deep copy of itself

```
ET_Iterator_Impl *Pre_Order_ET_Iterator_Impl::clone()
{
  return new Pre_Order_ET_Iterator_Impl(*this);
}
```

Prototype GoF Object Creational

Consequences

+ Can add & remove classes at runtime by cloning them as needed
+ Reduced subclassing minimizes need for lexical dependencies at run-time
– Every class that used as a prototype must itself be instantiated
– Classes that have circular references to other classes cannot really be cloned

Prototype GoF Object Creational

Consequences

+ Can add & remove classes at runtime by cloning them as needed
+ Reduced subclassing minimizes need for lexical dependencies at run-time
– Every class that used as a prototype must itself be instantiated
– Classes that have circular references to other classes cannot really be cloned

Implementation

- Use prototype manager
- Shallow vs. deep copies
- Initializing clone internal state within a uniform interface

Prototype GoF Object Creational

Consequences

+ Can add & remove classes at runtime by cloning them as needed
+ Reduced subclassing minimizes need for lexical dependencies at run-time
– Every class that used as a prototype must itself be instantiated
– Classes that have circular references to other classes cannot really be cloned

Implementation

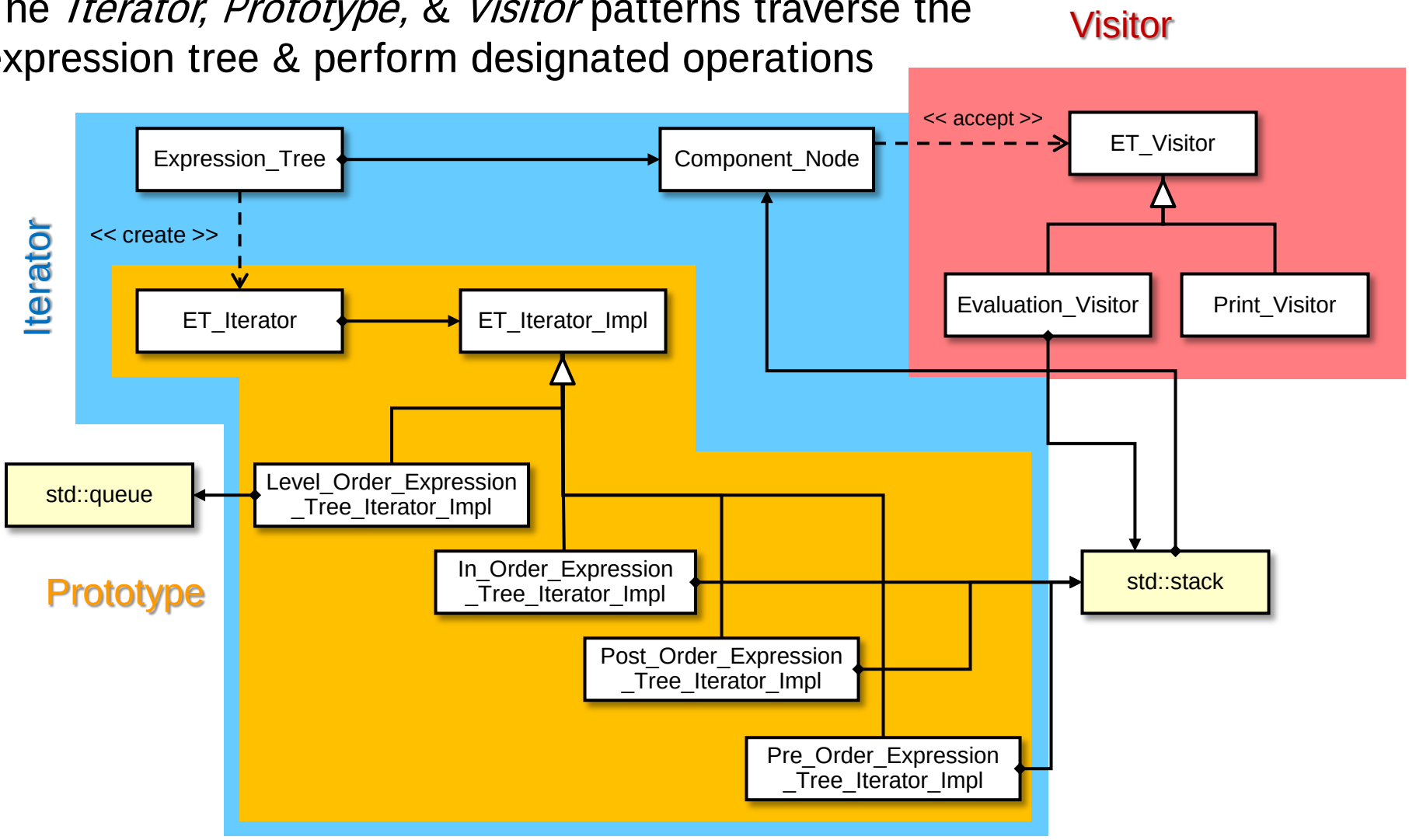
- Use prototype manager
- Shallow vs. deep copies
- Initializing clone internal state within a uniform interface

Known Uses

- The first widely known application of the Prototype pattern in an object-oriented language was in ThingLab
- Jim Coplien describes idioms related to the Prototype pattern for C++ & gives many examples & variations
- Etgdb debugger for ET++
- The music editor example is based on the Unidraw drawing framework

Summary of Tree Traversal Patterns

The *Iterator, Prototype,* & *Visitor* patterns traverse the expression tree & perform designated operations



These patterns allow adding new operations without affecting tree structure