The Factory Method Pattern

Other Considerations

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Factory Method* pattern can be applied to extensibly create variabilities in the expression tree processing app.
- Understand the structure & functionality of the *Factory Method* pattern.
- Know how to implement the *Factory Method* pattern in C++.
- Be aware of other considerations when applying the *Factory Method* pattern.

Douglas C. Schmidt

Other Considerations of the Factory Method Pattern

Consequences

+ *Decoupling*

- Clients are more flexible since they needn't specify the class name of the concrete class & the details of its creation.

Instead of:

```
User_Command command =
  new Print_Command();
```

Use:

```
User_Command command
   = command_factory_.
       make_command
          ("print"));
```

where **`userCommand_Factory`** is an instance of **`User_Command_Factory`**

Consequences

+ *Decoupling*

- Clients are more flexible since they needn't specify the class name of the concrete class & the details of its creation.

Hard-codes a lexical dependency on `Print_Command`

Instead of:

```
User_Command_Impl command =
  new Print_Command();
```

Use:

```
User_Command command
   = command_factory_.
       make_command
          ("print");
```

where `command_factory_` is an instance of `User_Command_Factory`

Consequences

\+ *Decoupling*

- Clients are more flexible since they needn't specify the class name of the concrete class & the details of its creation.

Instead of:

```
User_Command_Impl command =
  new Print_Command();
```

Use:

No lexical dependency on any concrete class

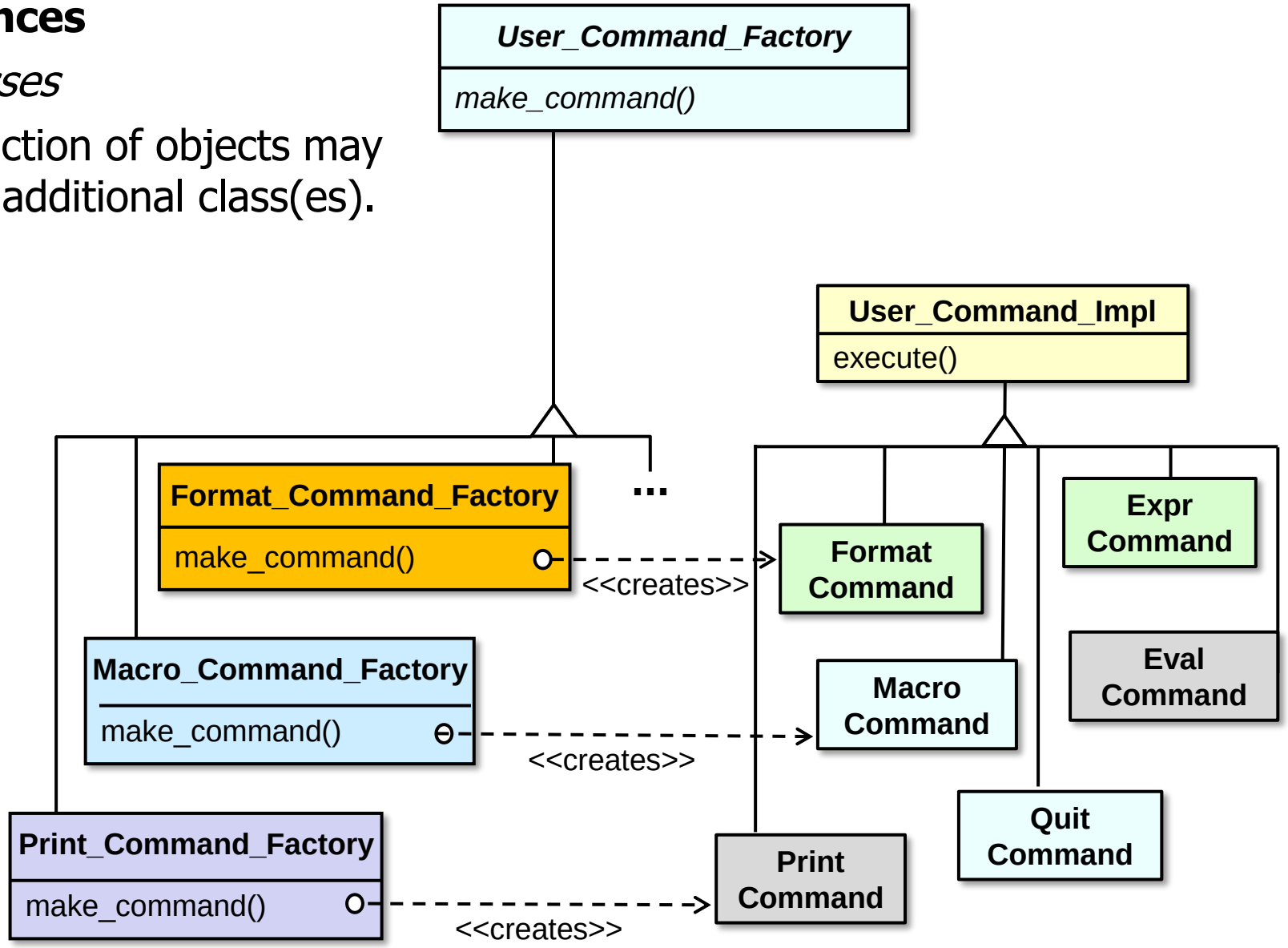
```
User_Command command
  = command_factory_.
      make_command
        ("print");
```

where `command_factory_` is an instance of `User_Command_Factory`

Consequences

– *More classes*

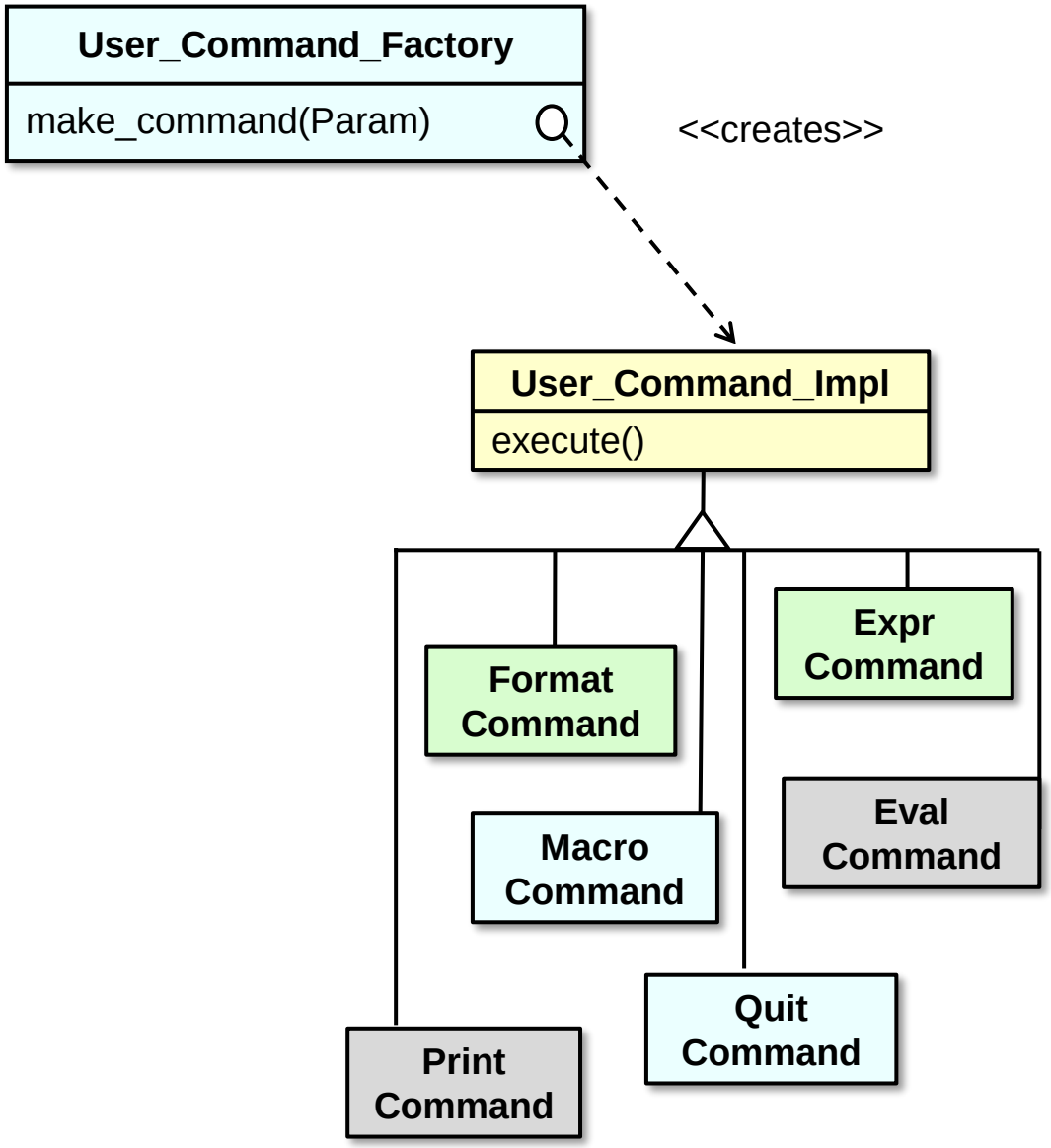
- Construction of objects may require additional class(es).



Consequences

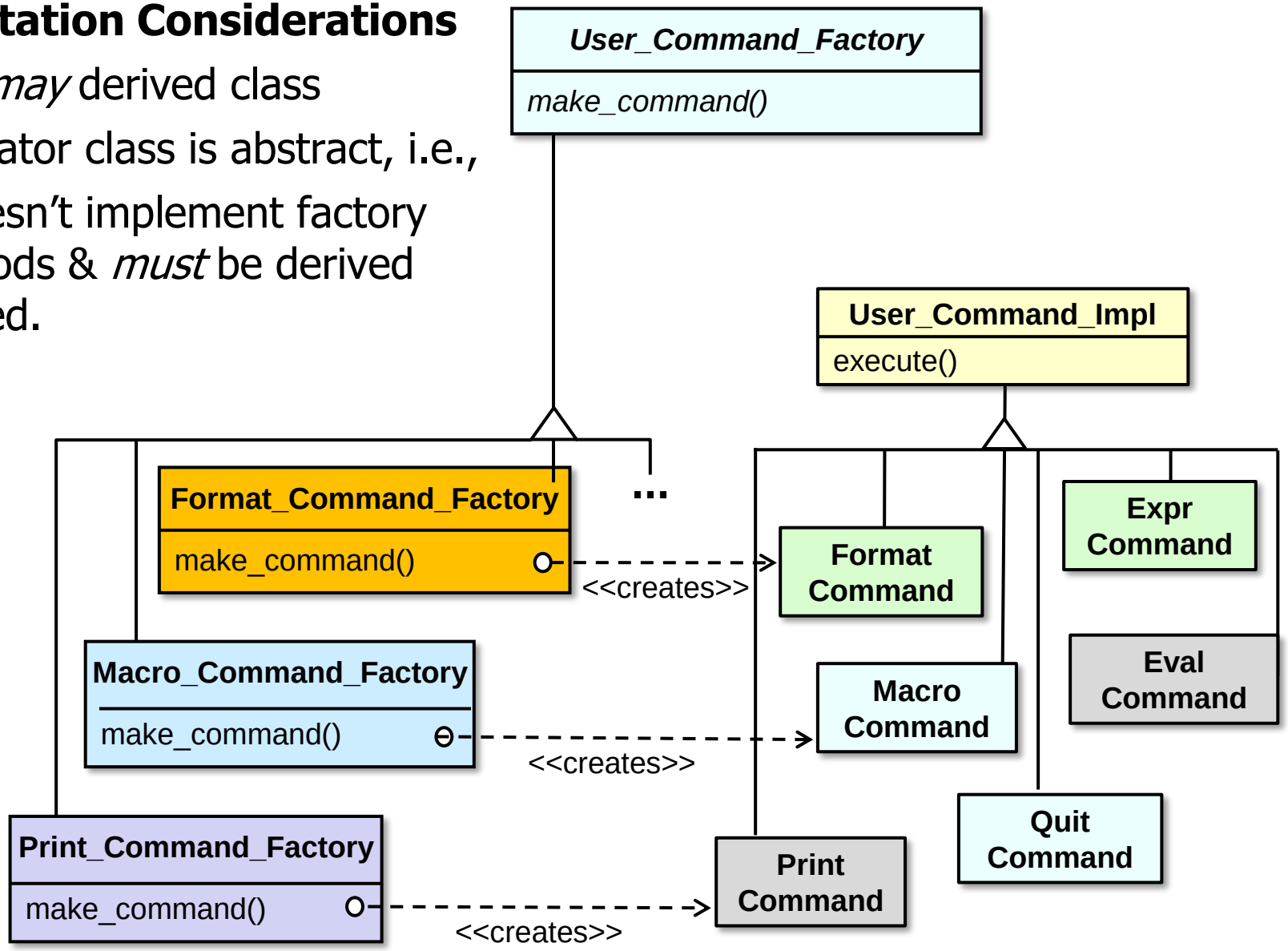
– *More classes*

- Construction of objects may require additional class(es).
- An alternative is to pass a param to the Creator super class factory method.



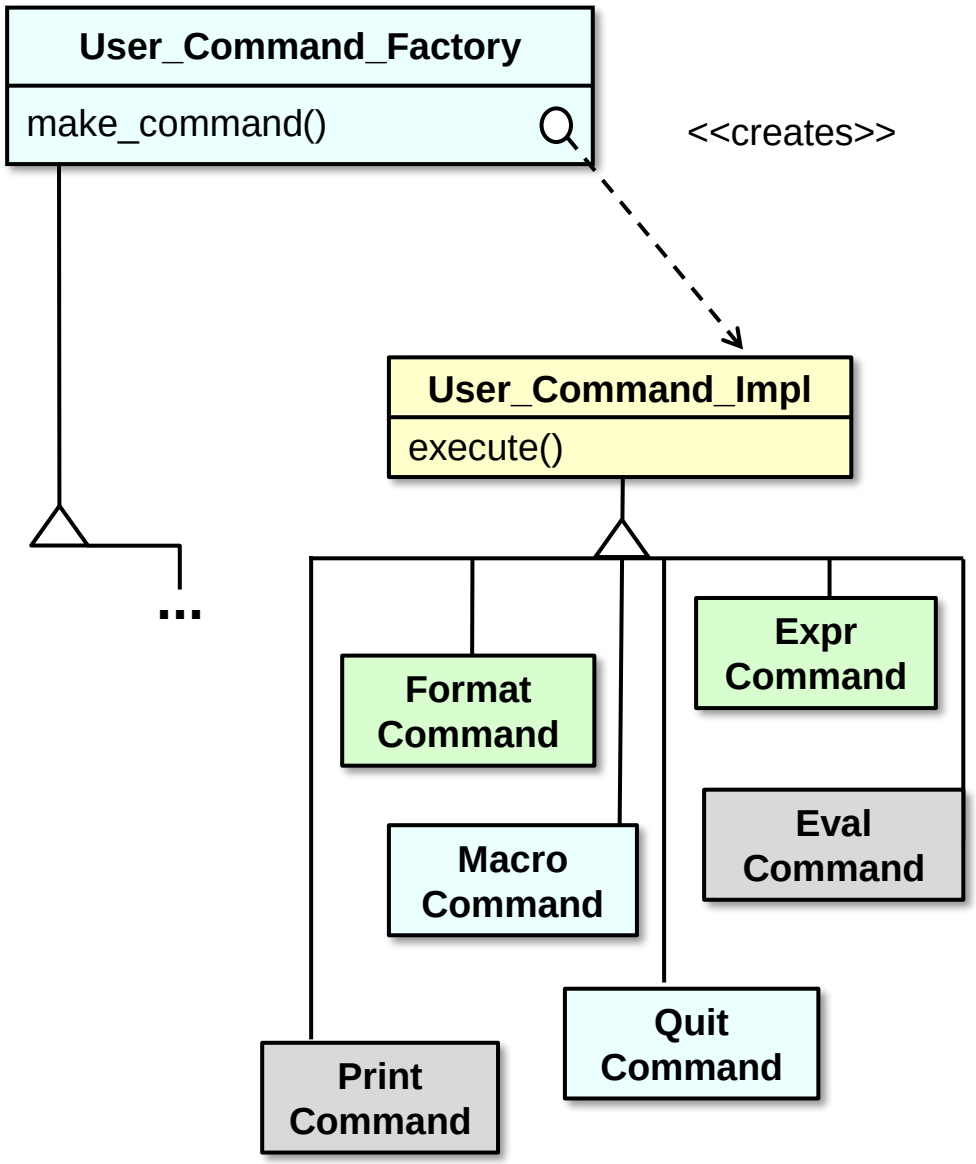
Implementation Considerations

- *Must* vs. *may* derived class
  - The creator class is abstract, i.e.,
    - It doesn't implement factory methods & *must* be derived classed.



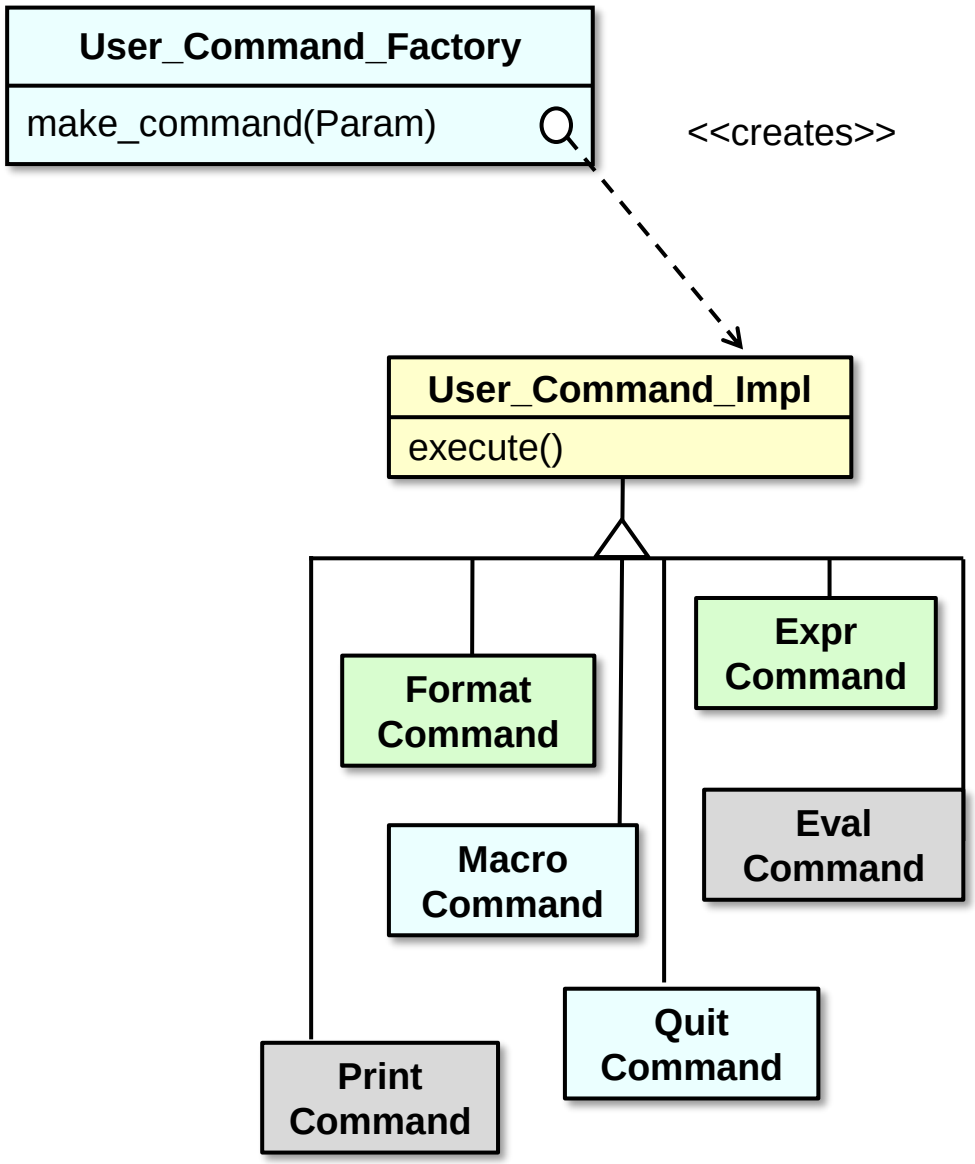
Implementation Considerations

- *Must* vs. *may* derived class
  - The creator class is abstract.
  - The creator class is concrete, i.e.,
    - It provides a default factory method & *may* be derived classed.



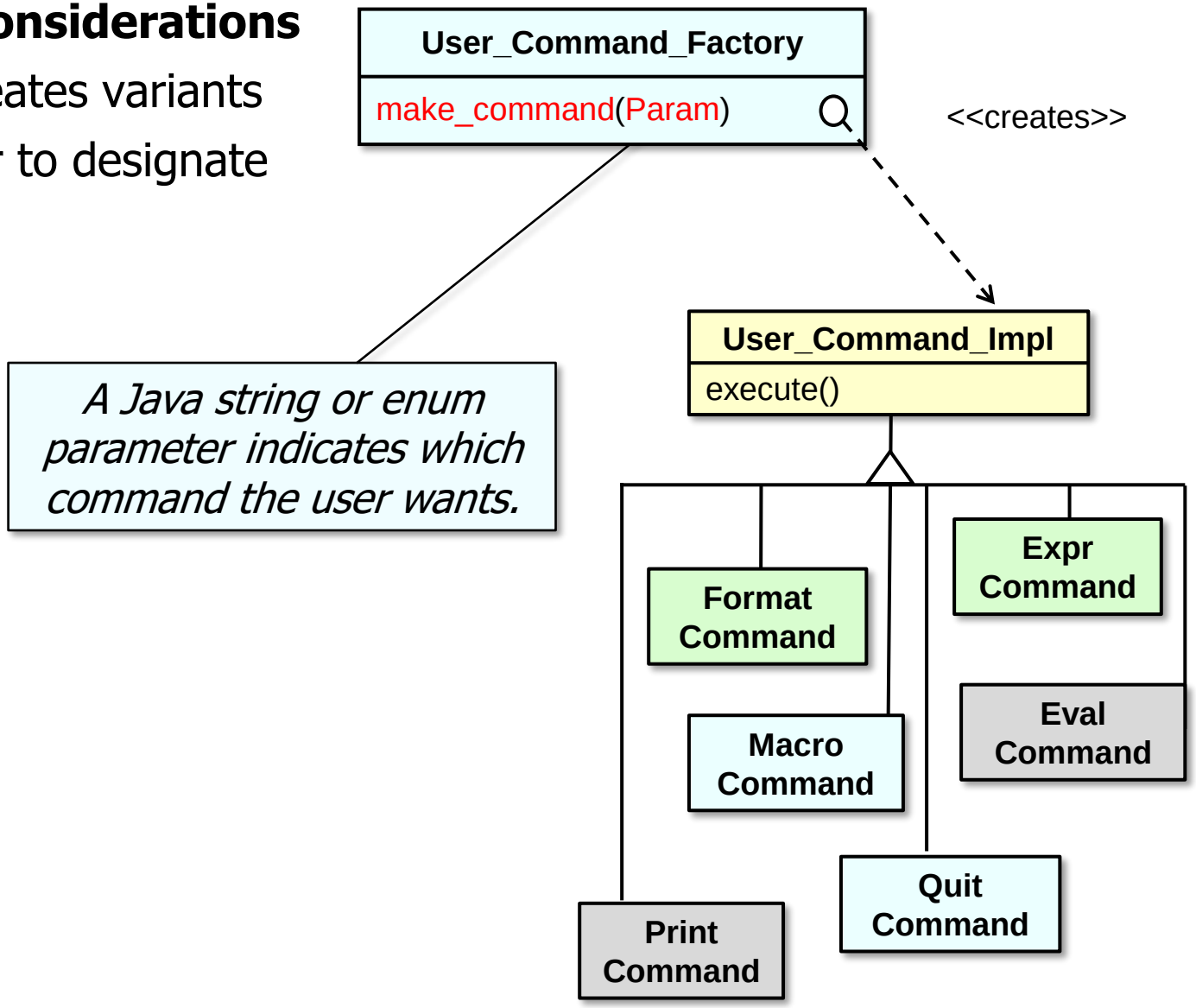
Implementation Considerations

- Factory method creates variants
  - Pass a parameter to designate the variant.



Implementation Considerations

- Factory method creates variants
  - Pass a parameter to designate the variant.



A string is more flexible, whereas an enum is more type-safe.

Implementation Considerations

- Constructor references in modern Java may reduce the tedium of creating Product derived classes

```
class ShapeFactory {
  Map<string, Supplier<Shape>> map =
    new std::map<>() {{
      put("CIRCLE", Circle::new);
      put("RECTANGLE", Rectangle::new);
      ...
  }};

  Shape getShape(string shape) {
    Supplier<Shape> shape = map.get(shape.toUpperCase());
    if (shape != null)
      return shape.get();
    throw new IllegalArgumentException
      ("No such shape " + shape.toUpperCase());
    }
  }
}
```

See dzone.com/articles/factory-pattern-using-lambda-expression-in-java-8

Implementation Considerations

- Constructor references in modern Java may reduce the tedium of creating Product derived classes

```
class ShapeFactory {
  Map<string, Supplier<Shape>> map =
    new std::map<>() {{
      put("CIRCLE", Circle::new);
      put("RECTANGLE", Rectangle::new);
      ...
  }};

  Shape getShape(string shape) {
    Supplier<Shape> shape = map.get(shape.toUpperCase());
    if (shape != null)
      return shape.get();
    throw new IllegalArgumentException
      ("No such shape " + shape.toUpperCase());
    }
  }
}
```

Constructor references can be used to create desired shapes.

Implementation Considerations

- Constructor references in modern Java may reduce the tedium of creating Product derived classes

```
class ShapeFactory {
  Map<string, Supplier<Shape>> map =
    new std::map<>() {{
      put("CIRCLE", Circle::new);
      put("RECTANGLE", Rectangle::new);
      ...
  }};

  Shape getShape(string shape) {
    Supplier<Shape> shape = map.get(shape.toUpperCase());
    if (shape != null)
      return shape.get();
    throw new IllegalArgumentException
      ("No such shape " + shape.toUpperCase());
    }
  }
}
```

Get & create the requested Shape derived class

Implementation Considerations

- Constructor references in modern Java may reduce the tedium of creating Product derived classes

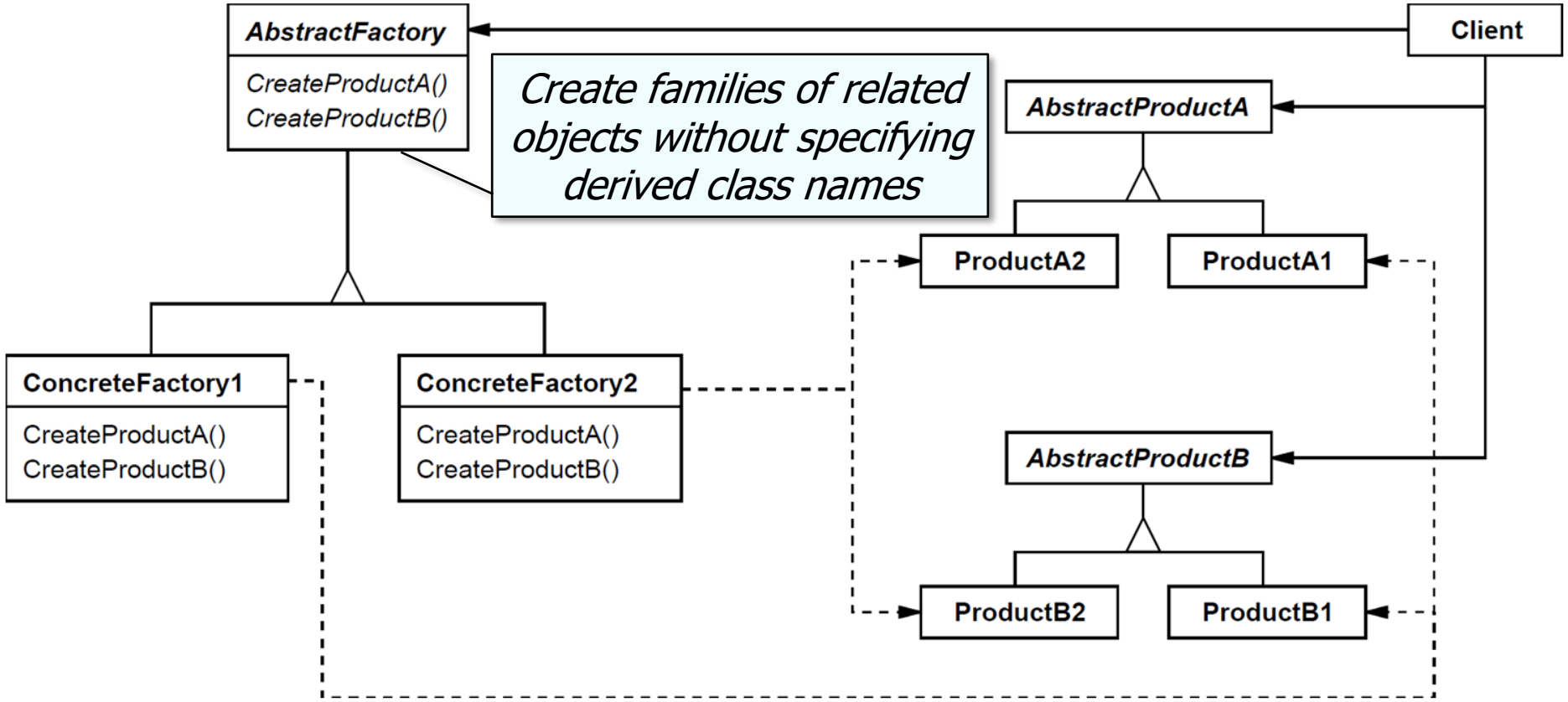
```
class ShapeFactory {
  Map<string, Supplier<Shape>> map =
    new std::map<>() {{
      put("CIRCLE", Circle::new);
      put("RECTANGLE", Rectangle::new);
      ...
  }};

  Shape getShape(string shape) {
    Supplier<Shape> shape = map.get(shape.toUpperCase());
    if (shape != null)
      return shape.get();
    throw new IllegalArgumentException
      ("No such shape " + shape.toUpperCase());
    }
  }
}
```

Doesn't scale if getShape() takes multiple arguments to pass to Shape constructors

Implementation Considerations

- Apply *Abstract Factory* if many semantically-consistent factory methods needed



See en.wikipedia.org/wiki/Abstract_factory_pattern

Known uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- BREW feature phone frameworks
- The ACE ORB (TAO)
- **`iterator()`** factory method in the Java **`Collection`** interface

iterator

```
Iterator<E> iterator()
```

Returns an iterator over the elements in this collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee).

Specified by:

`iterator` in interface `Iterable<E>`

Returns:

an Iterator over the elements in this collection

See docs.oracle.com/javase/8/docs/api/java/util/Collection.html#iterator

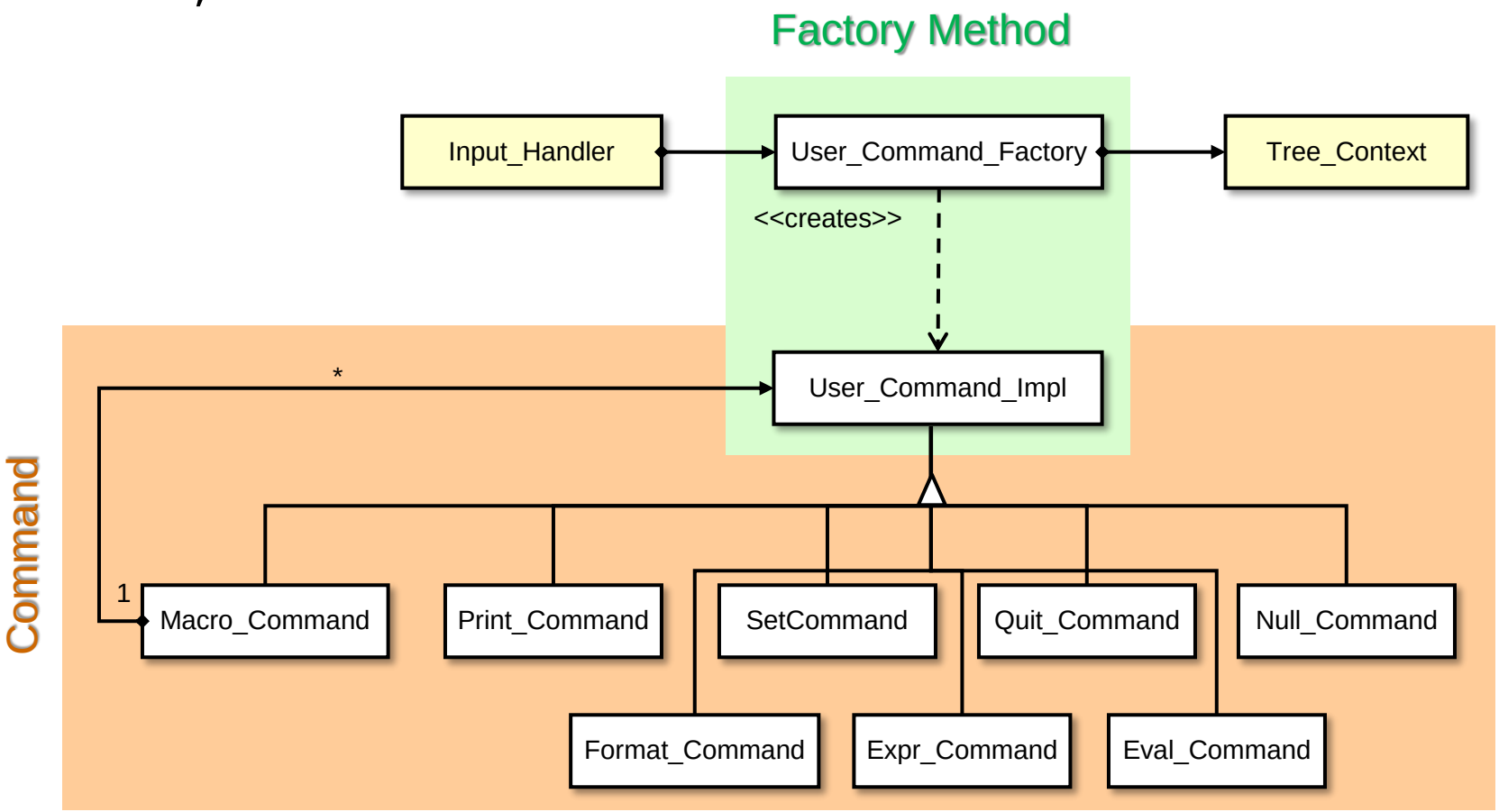
Known uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- BREW feature phone frameworks
- The ACE ORB (TAO)
- `iterator()` factory method in the Java `Collection` interface
- The begin() & end() factory methods in C++ STL containers

Iterators	
`begin()` `cbegin()`	returns an iterator to the beginning
`end()` `cend()`	returns an iterator to the end
`rbegin()` `crbegin()`	returns a reverse iterator to the beginning
`rend()` `crend()`	returns a reverse iterator to the end

Summary of the Factory Method Pattern

- *Factory Method* enables extensible creation of variabilities, such as iterators, commands, & visitors.



Factory Method decouples the creation of objects from their subsequent use.