Lecture_16 (3)

Wednesday, March 30, 2022 10:48 AM

BUTLER UNIVERSITY

SE 461 – Managing Software Development

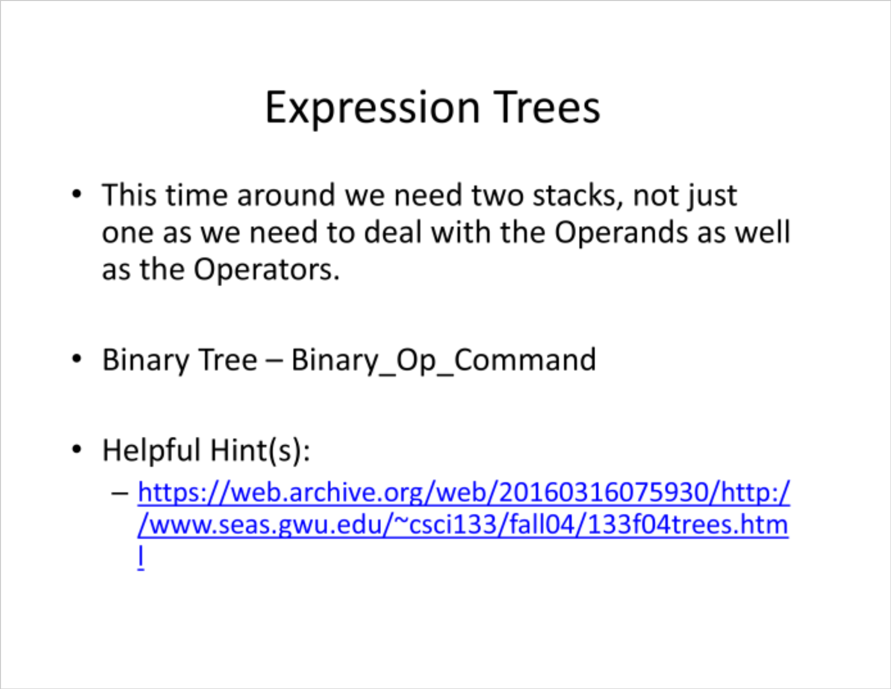
Lecture – 3/28/2022

Expression Trees

Expression Trees

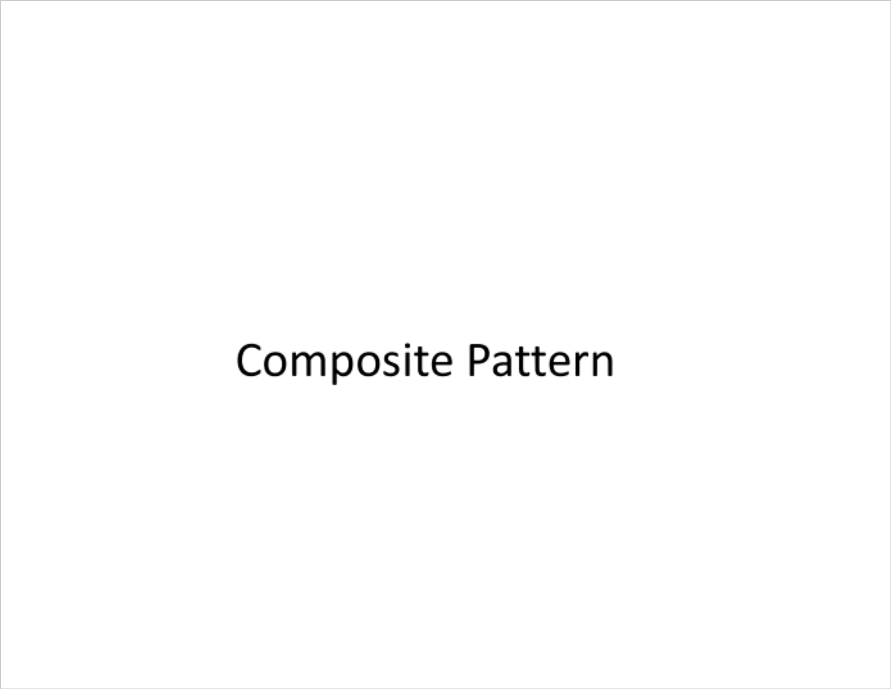
Expression Trees

- How do we construct an expression tree based upon the valid input of our program?
  - **Last Assignment:** Infix – Postfix Conversion
    - Same approach for the tree…
- The decision process for pushing and popping operators on and off the operator stack is exactly the same as we had in the Infix-Postfix Conversion….

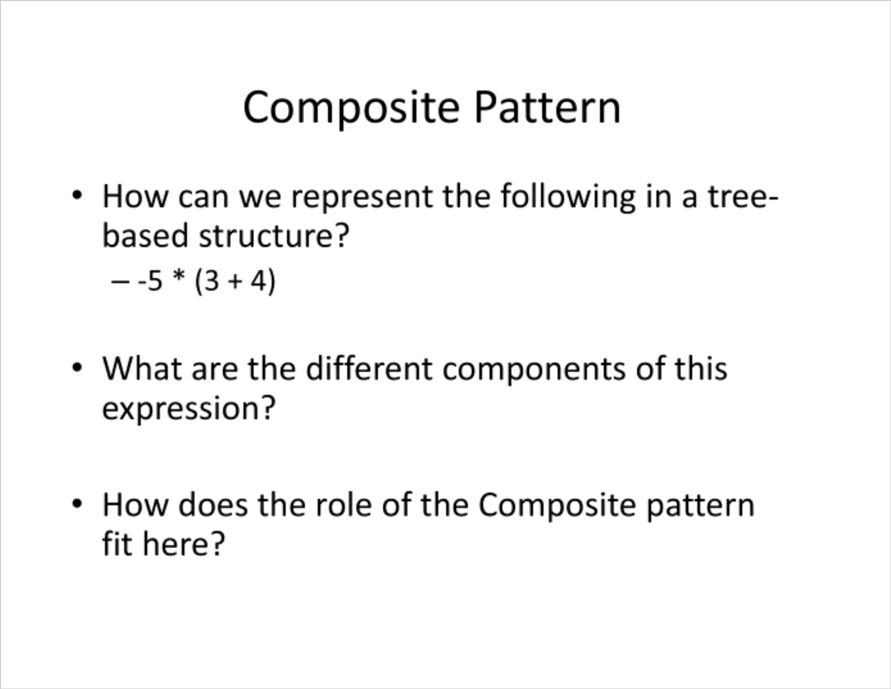


Expression Trees

- This time around we need two stacks, not just one as we need to deal with the Operands as well as the Operators.
- Binary Tree – Binary_Op_Command
- Helpful Hint(s):
  - https://web.archive.org/web/20160316075930/http://www.seas.gwu.edu/~csci133/fall04/133f04trees.html

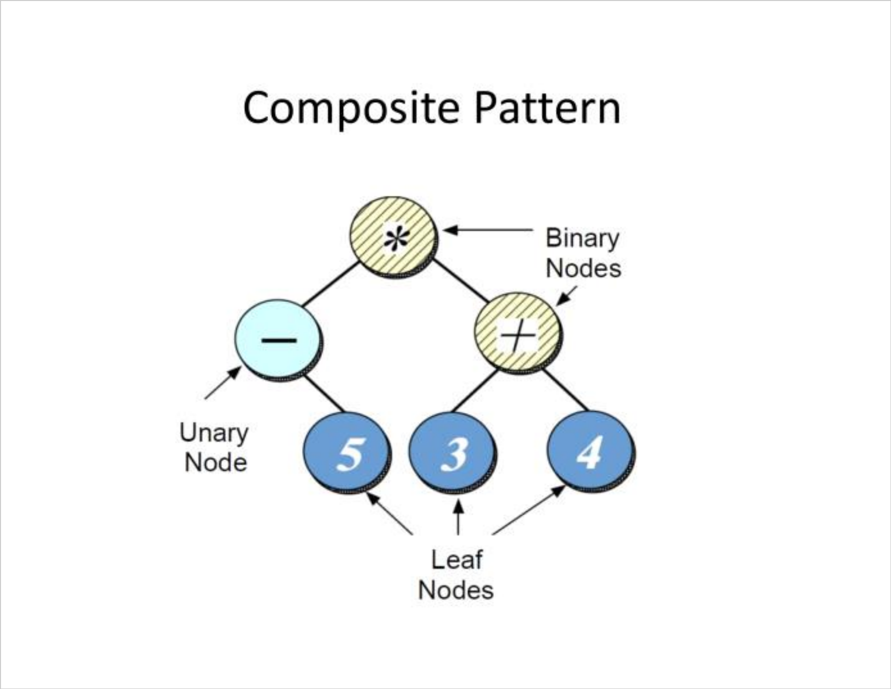


Composite Pattern

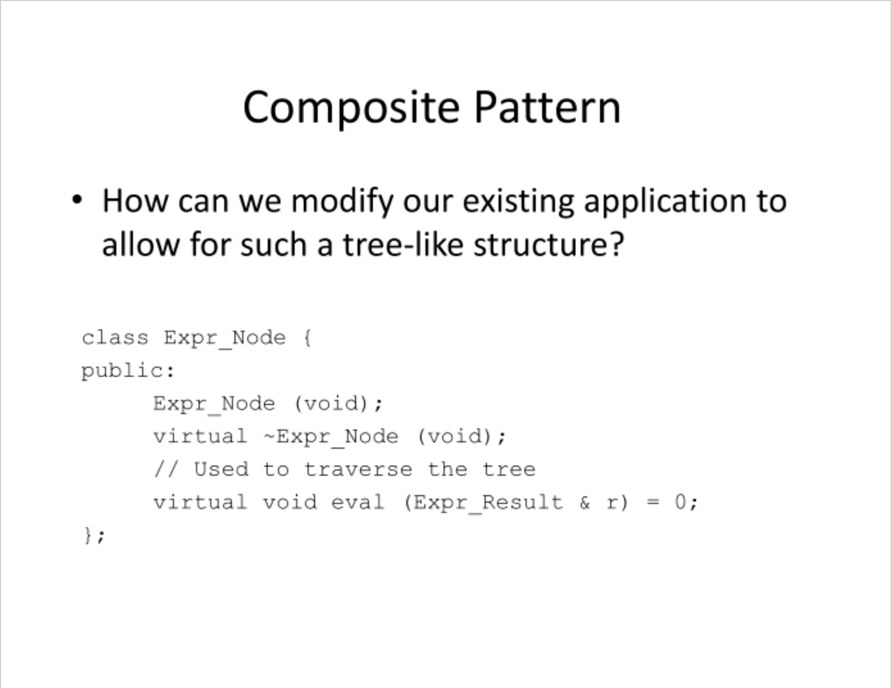


Composite Pattern

- How can we represent the following in a tree-based structure?
  - -5 * (3 + 4)
- What are the different components of this expression?
- How does the role of the Composite pattern fit here?



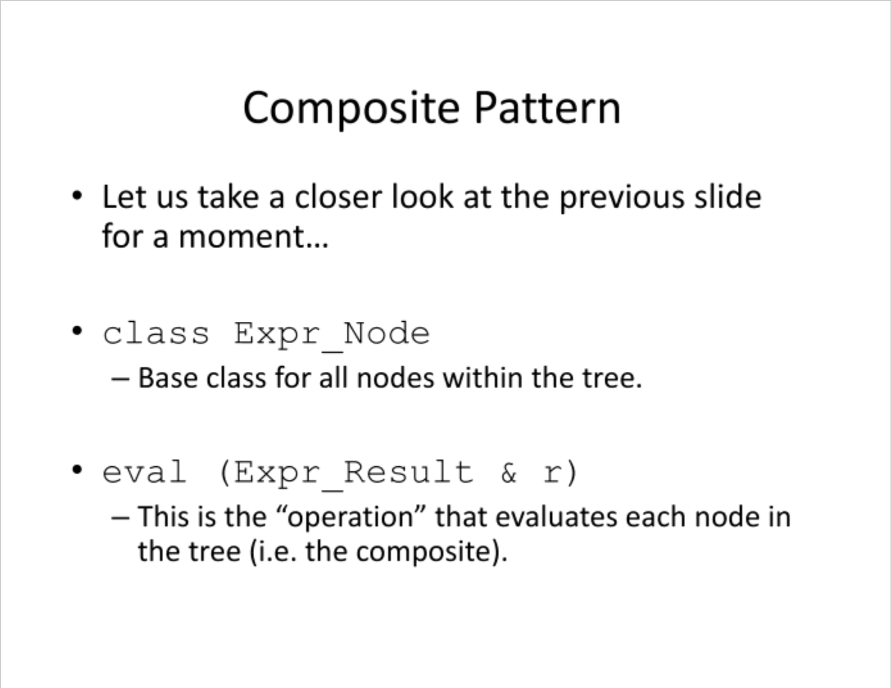
Composite Pattern



Composite Pattern

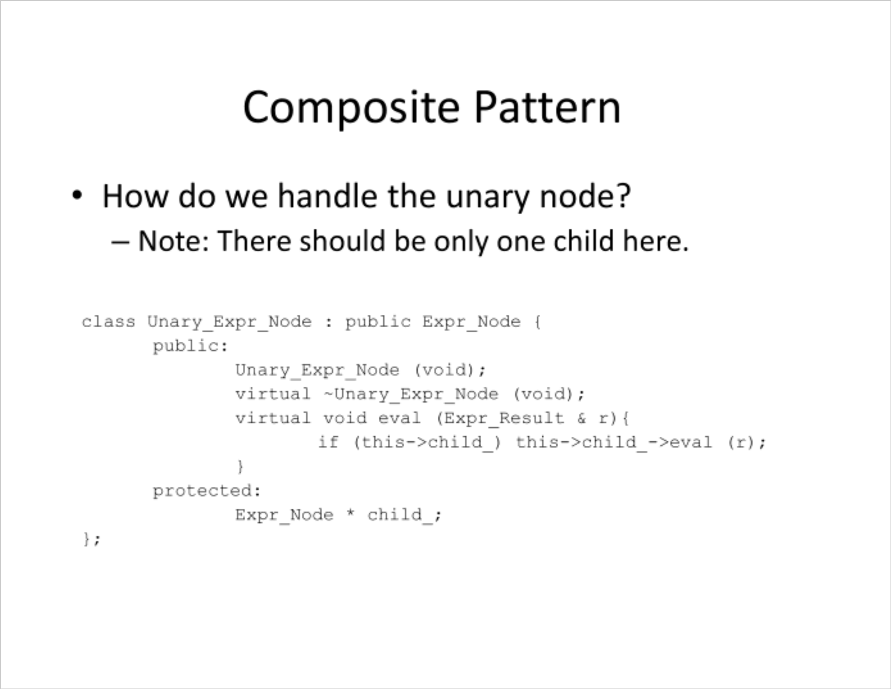
- How can we modify our existing application to allow for such a tree-like structure?

```
class Expr_Node {
public:
     Expr_Node (void);
     virtual ~Expr_Node (void);
     // Used to traverse the tree
     virtual void eval (Expr_Result & r) = 0;
};
```



Composite Pattern

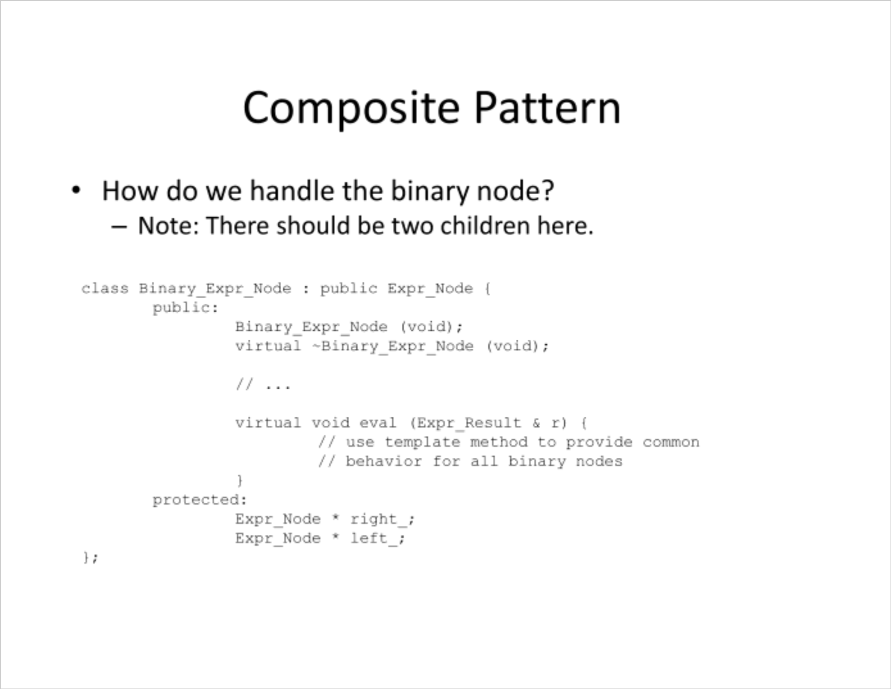
- Let us take a closer look at the previous slide for a moment…
- `class Expr_Node`
  - Base class for all nodes within the tree.
- `eval (Expr_Result & r)`
  - This is the “operation” that evaluates each node in the tree (i.e. the composite).



Composite Pattern

- How do we handle the unary node?
  - Note: There should be only one child here.

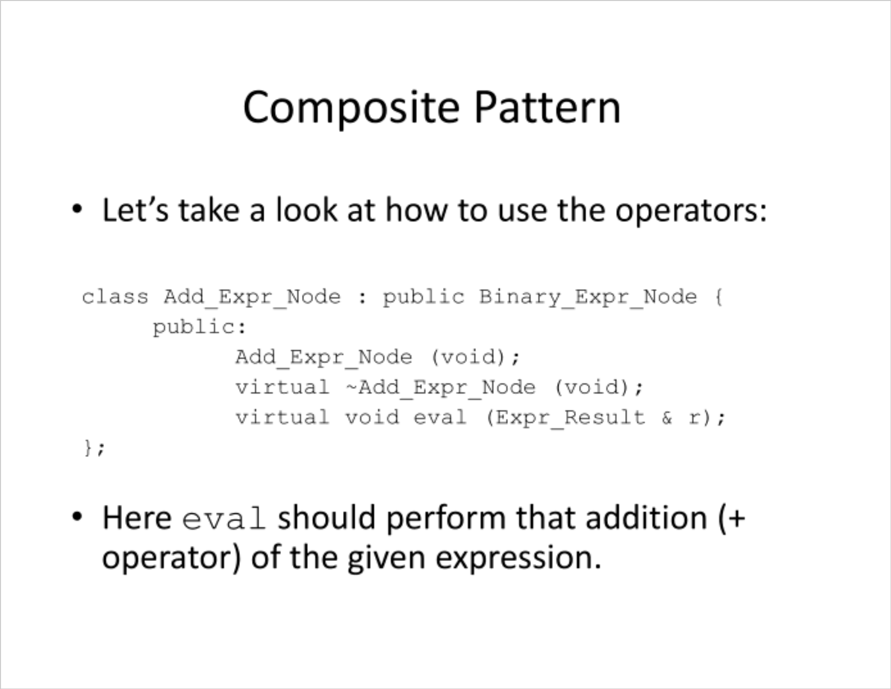
```
class Unary_Expr_Node : public Expr_Node {
      public:
            Unary_Expr_Node (void);
            virtual ~Unary_Expr_Node (void);
            virtual void eval (Expr_Result & r){
                  if (this->child_) this->child_->eval (r);
            }
      protected:
            Expr_Node * child_;
};
```



Composite Pattern

- How do we handle the binary node?
  - Note: There should be two children here.

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

            // ...

            virtual void eval (Expr_Result & r) {
                  // use template method to provide common
                  // behavior for all binary nodes
            }
      protected:
            Expr_Node * right_;
            Expr_Node * left_;
};
```

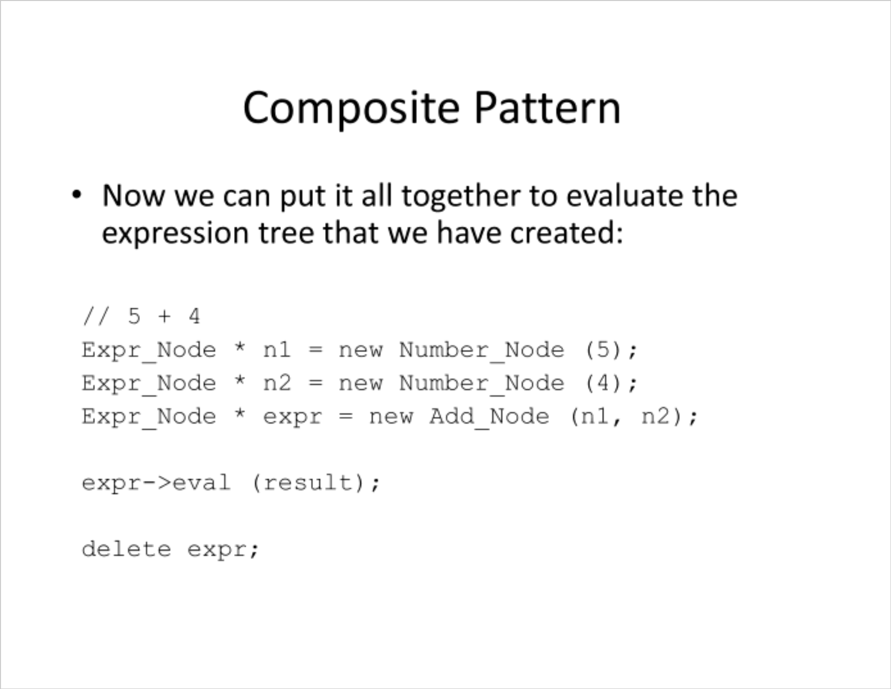


Composite Pattern

- Let’s take a look at how to use the operators:

```
class Add_Expr_Node : public Binary_Expr_Node {
     public:
           Add_Expr_Node (void);
           virtual ~Add_Expr_Node (void);
           virtual void eval (Expr_Result & r);
};
```

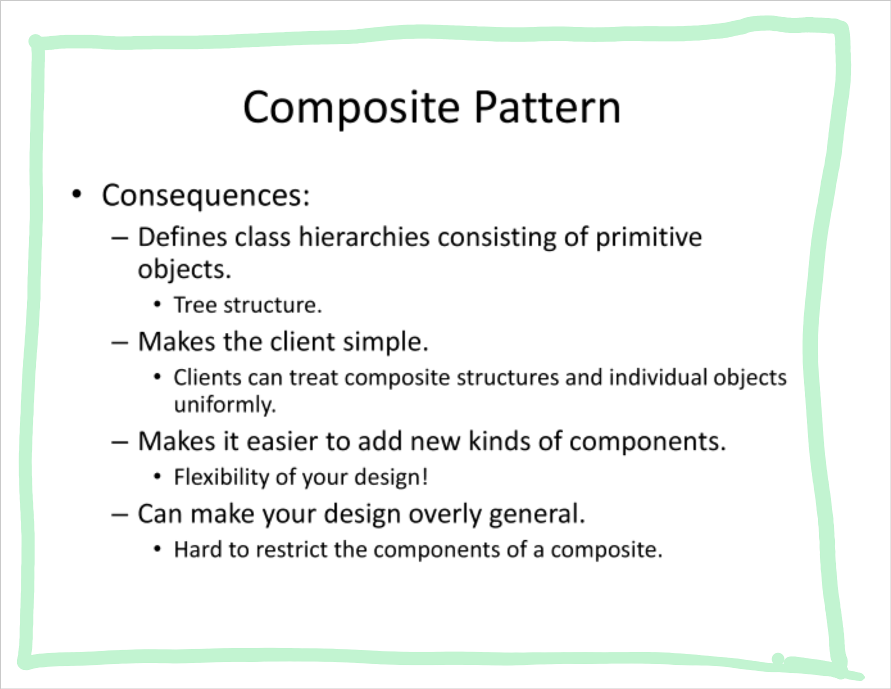
- Here `eval` should perform that addition (+ operator) of the given expression.



Composite Pattern

- Now we can put it all together to evaluate the expression tree that we have created:

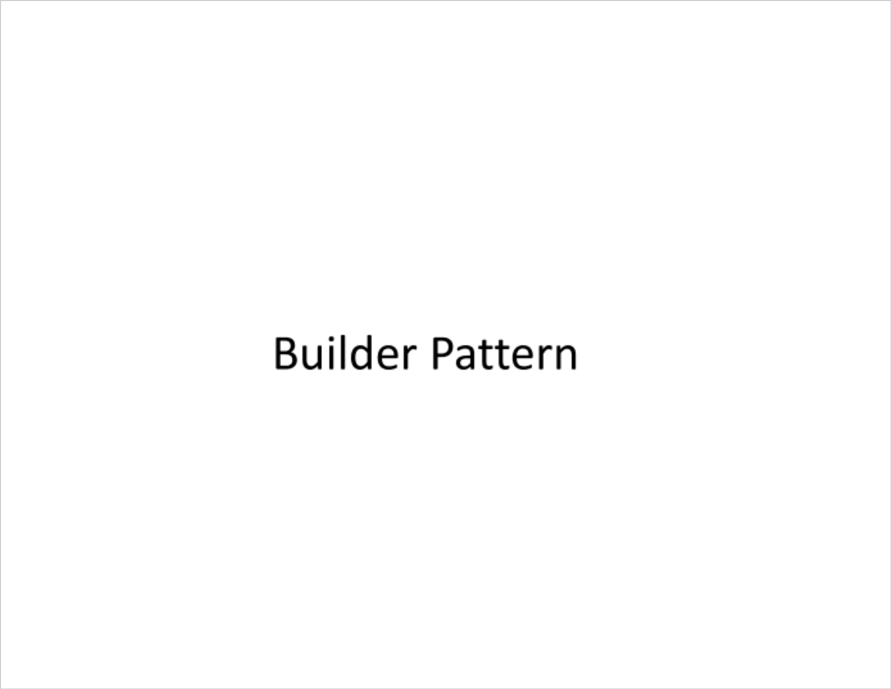
```
// 5 + 4
Expr_Node * n1 = new Number_Node (5);
Expr_Node * n2 = new Number_Node (4);
Expr_Node * expr = new Add_Node (n1, n2);

expr->eval (result);

delete expr;
```



Composite Pattern

- Consequences:
  - Defines class hierarchies consisting of primitive objects.
    - Tree structure.
  - Makes the client simple.
    - Clients can treat composite structures and individual objects uniformly.
  - Makes it easier to add new kinds of components.
    - Flexibility of your design!
  - Can make your design overly general.
    - Hard to restrict the components of a composite.

→ Abstraction vs Concretion



Builder Pattern

Builder Pattern

→ Builds something complex, while hiding the complexity from the user

- Pattern Classification:
  - Creational
- Problem:
  - Building a complex object but want to shield the client from the complexity.
- Solution:
  - Separate how we construct a complex object from its representation. This shields the client.

Builder Pattern - Motivation

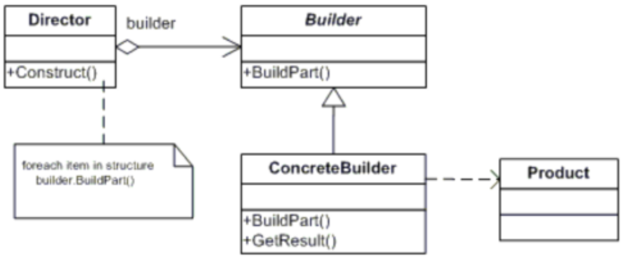
- Have you ever been in the following situation:
  - Need to build a complex object, but want to shield the client from the complexity of building the object.
    - e.g., Converting an infix expression to a binary tree.
  - Have many ways of building the same abstraction, but with different internal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

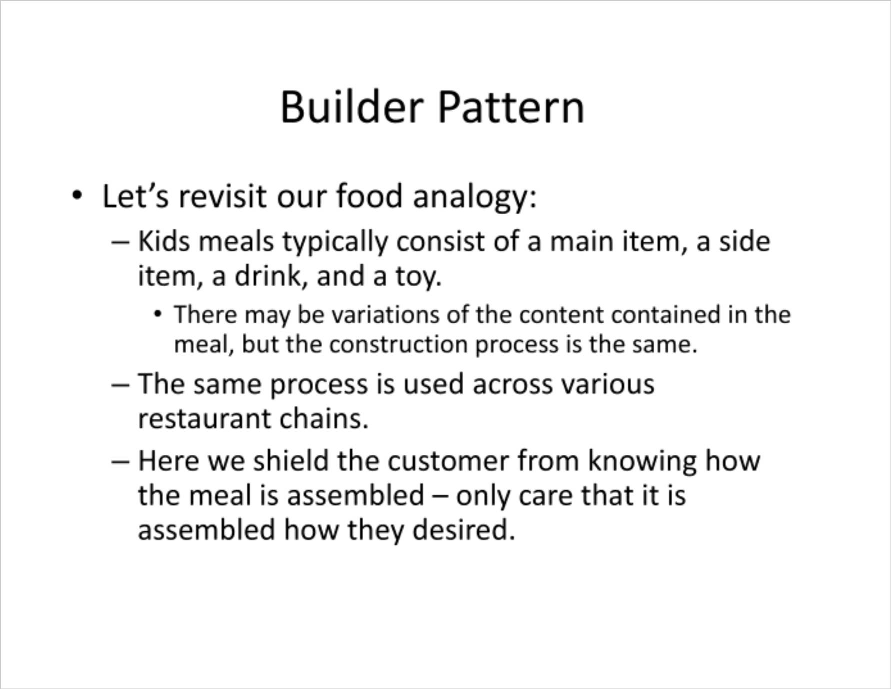
Builder Pattern

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

- How can we translate this analogy to the software domain?

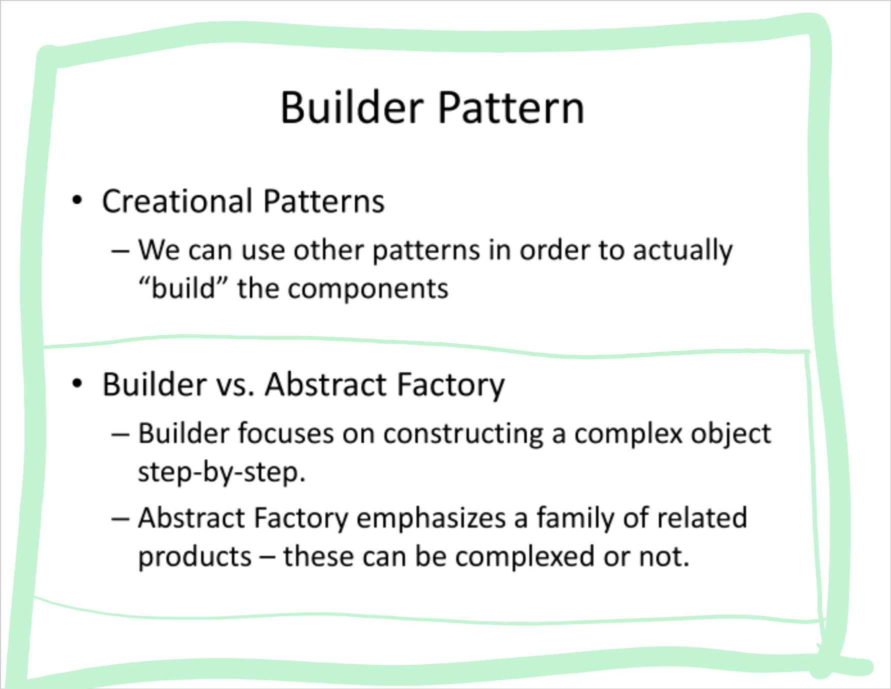
Builder Pattern





Builder Pattern

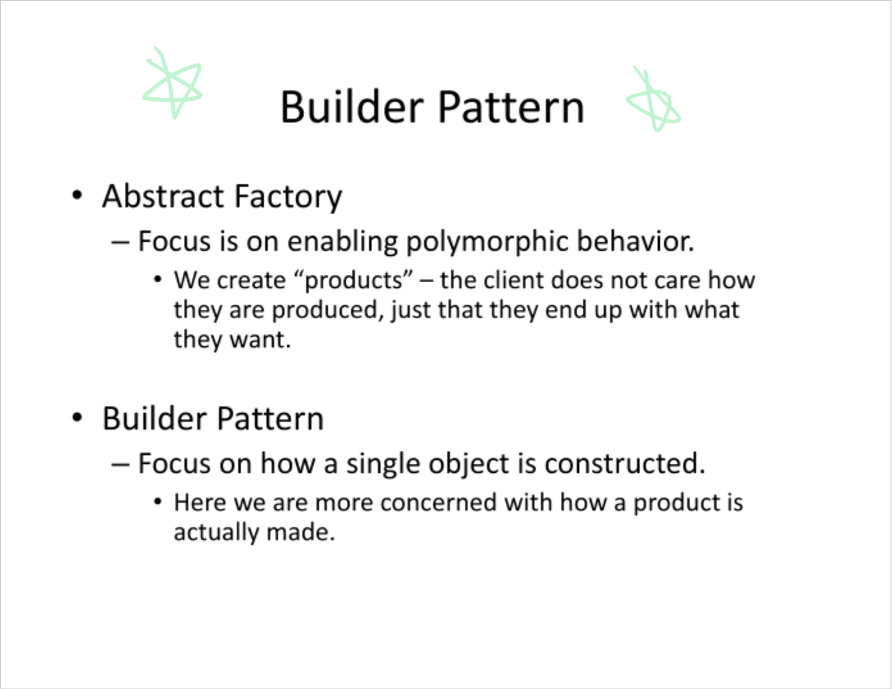
- Let's revisit our food analogy:
  - Kids meals typically consist of a main item, a side item, a drink, and a toy.
    - There may be variations of the content contained in the meal, but the construction process is the same.
  - The same process is used across various restaurant chains.
  - Here we shield the customer from knowing how the meal is assembled – only care that it is assembled how they desired.



Builder Pattern

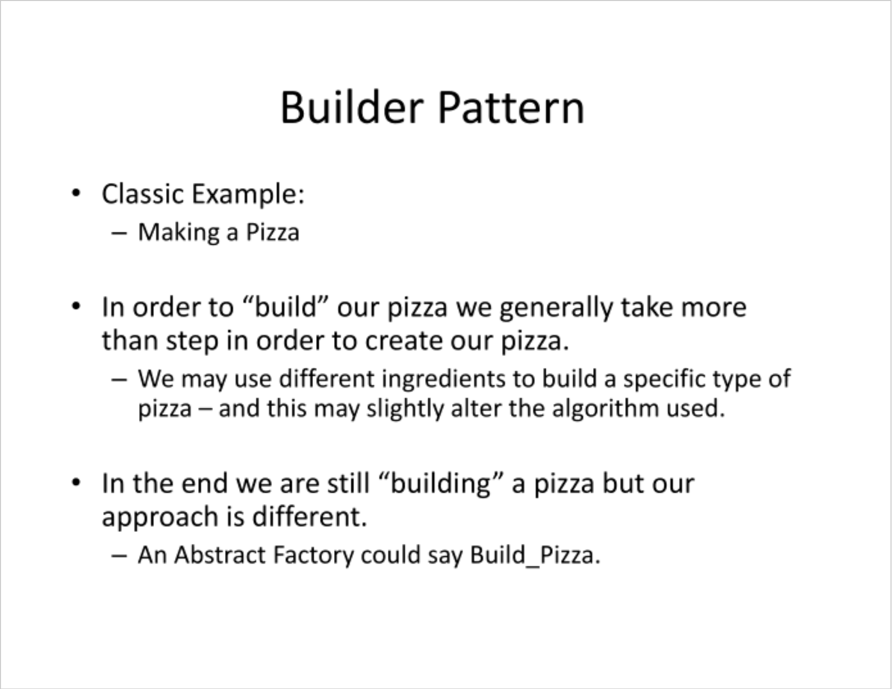
- Creational Patterns
  - We can use other patterns in order to actually "build" the components
- Builder vs. Abstract Factory
  - Builder focuses on constructing a complex object step-by-step.
  - Abstract Factory emphasizes a family of related products – these can be complexed or not.

You gotta have the Structure to create the parts



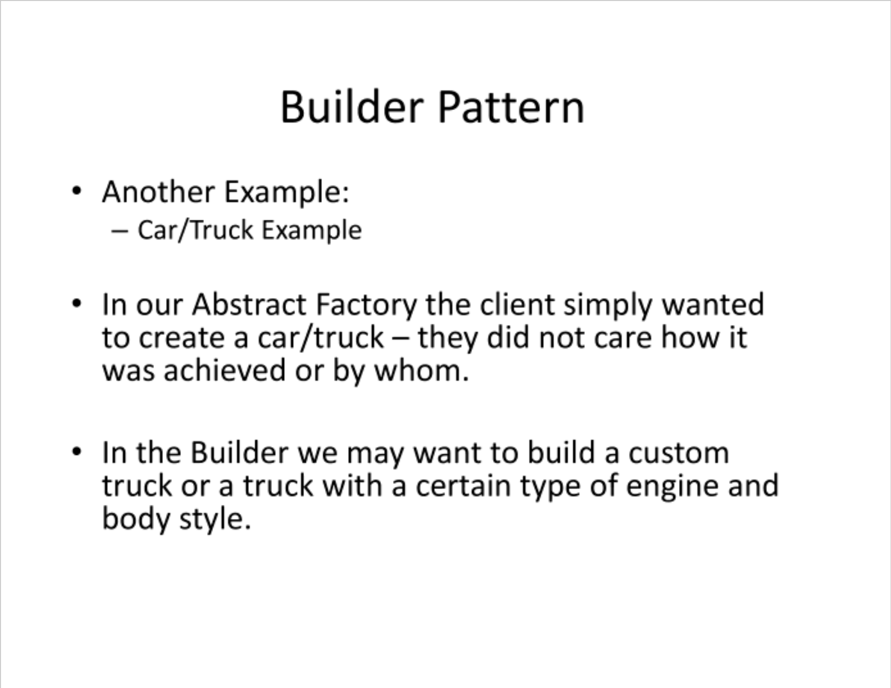
Builder Pattern

- Abstract Factory
  - Focus is on enabling polymorphic behavior.
    - We create “products” – the client does not care how they are produced, just that they end up with what they want.
- Builder Pattern
  - Focus on how a single object is constructed.
    - Here we are more concerned with how a product is actually made.



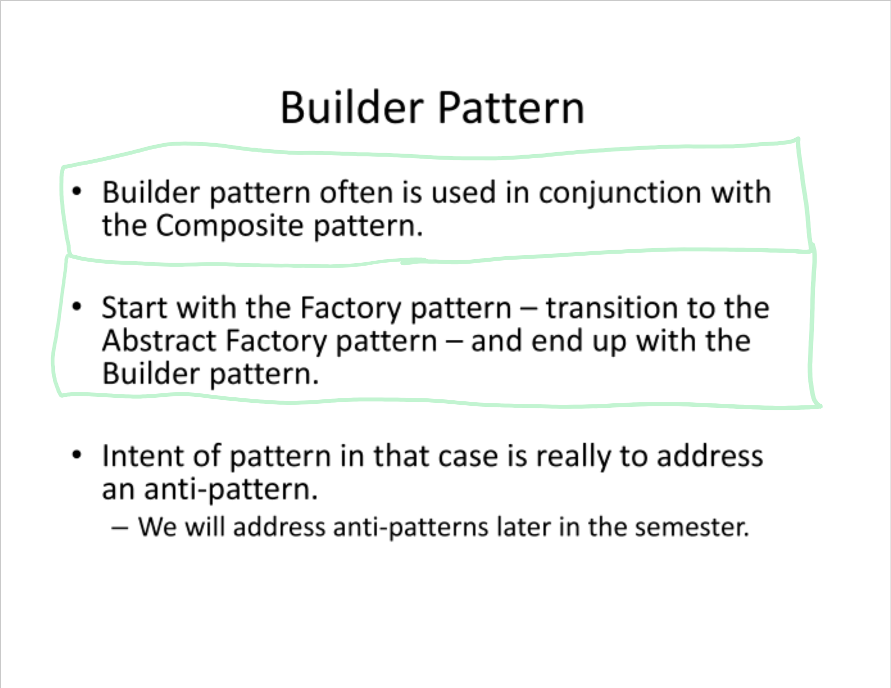
Builder Pattern

- Classic Example:
  - Making a Pizza
- In order to “build” our pizza we generally take more than step in order to create our pizza.
  - We may use different ingredients to build a specific type of pizza – and this may slightly alter the algorithm used.
- In the end we are still “building” a pizza but our approach is different.
  - An Abstract Factory could say Build_Pizza.



Builder Pattern

- Another Example:
  - Car/Truck Example
- In our Abstract Factory the client simply wanted to create a car/truck – they did not care how it was achieved or by whom.
- In the Builder we may want to build a custom truck or a truck with a certain type of engine and body style.



Builder Pattern

- Builder pattern often is used in conjunction with the Composite pattern.
- Start with the Factory pattern – transition to the Abstract Factory pattern – and end up with the Builder pattern.
- Intent of pattern in that case is really to address an anti-pattern.
  - We will address anti-patterns later in the semester.