The Bridge Pattern

Motivating Example

Douglas C. Schmidt
Learning Objectives in This Lesson

- Recognize how the *Bridge* pattern can be applied to make the expression tree structure easier to access & evolve transparently.
Motivating the Need for the Bridge Pattern in the Expression Tree App

Douglas C. Schmidt
**Purpose:** Decouple the expression tree programming API from its behavior & implementation to enable transparent extensibility.

*Bridge* minimizes coupling between clients, abstractions, & implementations.
Context: OO Expression Tree Processing App

- The app needs to run in a range of design-time & runtime environments
Context: OO Expression Tree Processing App

- The app needs to run in a range of design-time & runtime environments, e.g.,
- Mobile devices with limited memory & processing power
The app needs to run in a range of design-time & runtime environments, e.g.,

- Mobile devices with limited memory & processing power
- Laptops & desktops with more abundant resources
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks.
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks.
- Suboptimal implementations for a given context

```java
Component_Node node =
    new Composite_Add_Node
        (new Leaf_Node(3),
         new Leaf_Node(4));
```
• Tightly coupling app components to a particular environment has drawbacks.

• Suboptimal implementations for a given context

```
Component_Node node =
    new Composite_Add_Node
    (new Leaf_Node(3),
     new Leaf_Node(4));
```

VS.

```
Component_Node node =
    new Tree_Node
    ('+',
     new Tree_Node(3),
     new Tree_Node(4));
```
Tightly coupling app components to a particular environment has drawbacks.

Suboptimal implementations for a given context

Different implementations have different time/space trade-offs

We should be able to change implementations without breaking client code.
**Problem: Minimizing Impact of Variability**

- Tightly coupling app components to a particular environment has drawbacks.
  - Suboptimal implementations for a given context
  - Hard to change services transparently

![Diagram of nodes and node types](image)
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks.
  - Suboptimal implementations for a given context
  - Hard to change services transparently, e.g.:
  - Want to transparently add instrumentation to expression tree operations
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks.
  - Suboptimal implementations for a given context
  - Hard to change services transparently, e.g.,
    - Want to transparently add instrumentation to expression tree operations
    - Want to transparently add synchronization to expression tree methods
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks.
  - Suboptimal implementations for a given context
  - Hard to change services transparently, e.g.,
    - Want to transparently add instrumentation to expression tree operations
    - Want to transparently add synchronization to expression tree methods
    - ...

We should be able to enhance the service without breaking its implementation.
Solution: Separate Abstraction & Implementation

• Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations.
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations.

Variations in expression tree implementations
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations.

Variations in what service is provided by an expression tree
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations.
- Client calls to the abstraction are forwarded to the corresponding implementor subclass.

```
Expression_Tree
  accept(Visitor &v)
  ...

Component_Node
  accept(Visitor &v)

Leaf_Node
  accept(Visitor &v)
  ...

... Solution: Separate Abstraction & Implementation ...
```

```
... mRoot
... mRoot.accept(v);

... Solution: Separate Abstraction & Implementation ...
```
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations.
- Client calls to the abstraction are forwarded to the corresponding implementor subclass.
- Subclass the abstraction class to enable different services without affecting the implementor hierarchy.

```cpp
Expression_Tree
accept(Visitor &v)
...

Synchronized
Expression_Tree
accept(Visitor &v)

lock_guard<mutex>
guard(mMutex);
mRoot.accept(v);

Component_Node
accept(Visitor &v)

Leaf_Node
accept(Visitor &v)
...

...
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations.
- Client calls to the abstraction are forwarded to the corresponding implementor subclass.
- Subclass the abstraction class to enable different services without affecting the implementor hierarchy.
Expression_Tree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

**Class methods**

```cpp
Expression_Tree( Component_Node *root)

bool is_null()
int item()

Expression_Tree left()
Expression_Tree right()

void accept(Visitor &visitor)
iterator begin(const string &traversal_order)
iterator end(const string &traversal_order)
```

See upcoming lessons on *Factory Method, Iterator, & Visitor* patterns.
Expression_Tree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

**Class methods**

Pass in the root of the implementor hierarchy

```cpp
Expression_Tree(Component_Node *root)
bool is_null()
int item()
Expression_Tree left()
Expression_Tree right()
void accept(Visitor &visitor)
iterator begin(const string &traversal_order)
iterator end(const string &traversal_order)
```
Expression_Tree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime.

**Class methods**

Forward to implementor hierarchy

```
Expression_Tree(Component_Node *root)

bool is_null()

int item()

Expression_Tree left()

Expression_Tree right()

void accept(Visitor &visitor)

iterator begin(const string &traversal_order)

iterator end(const string &traversal_order)
```
Expression_Tree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

**Class methods**

```cpp
Expression_Tree(Component_Node *root)
bool is_null()
int item()
Expression_Tree left()
Expression_Tree right()
void accept(Visitor &visitor)
iterator begin(const string &traversal_order)
iterator end(const string &traversal_order)
```

Plays essential role in the **Iterator & Visitor patterns.**

See upcoming lessons on “The Iterator Pattern” & “The Visitor Pattern.”
Expression_Tree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

Class methods

```cpp
Expression_Tree(Component_Node *root)
bool is_null()
int item()
Expression_Tree left()
Expression_Tree right()
void accept(Visitor &visitor)
iterator begin(const string &traversal_order)
iterator end(const string &traversal_order)
```

Factory methods creates iterators

See upcoming lessons on Factory Method, Iterator, & Strategy patterns.
Expression_Tree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

Class methods

```c
Expression_Tree (Component_Node *root)
bool is_null()
int item()
Expression_Tree left()
Expression_Tree right()
void accept (Visitor &visitor)
iterator begin (const string &traversal_order)
iterator end (const string &traversal_order)
```

- **Commonality**: provides a common interface for expression tree operations
- **Variability**: component nodes will vary depending on user input expressions; iterator behavior can vary; & expression tree itself can vary