Dependency Injection (DI) in C++

Introduction to Dependency Injection

Dependency Injection (DI) is a design pattern used to implement the Dependency Inversion Principle (DIP). It allows dependencies to be provided from external sources rather than being created within a class, improving modularity, testability, and maintainability.

Why Use Dependency Injection?

- **Loose Coupling:** Classes do not depend on concrete implementations but rather on abstractions.
- **Improved Testability:** Dependencies can be easily replaced with mock implementations for testing.
- **Better Maintainability:** Changes in dependencies do not affect dependent classes directly.

Example: Modeling Relationships Using Dependency Injection

We will modify our previous example to use Dependency Injection effectively.

Step 1: Define Relationship and Person Structures

```
enum class Relationship
{
  parent,
  child,
  sibling
};

struct Person
{
  string name;
};
```

Step 2: Introduce RelationshipBrowser Interface

To avoid tight coupling between `Research` and `Relationships`, we introduce an interface:

```
struct RelationshipBrowser
{
```

```
  virtual vector<Person> find_all_children_of(const string& name) = 0;
};
```

Step 3: Implement Relationships Class

The `Relationships` class now implements `RelationshipBrowser` and provides an actual implementation.

```
struct Relationships : RelationshipBrowser // low-level
{
  vector<tuple<Person, Relationship, Person>> relations;

  void add_parent_and_child(const Person& parent, const Person& child)
  {
    relations.push_back({parent, Relationship::parent, child});
    relations.push_back({child, Relationship::child, parent});
  }

  vector<Person> find_all_children_of(const string &name) override
  {
    vector<Person> result;
    for (auto&& [first, rel, second] : relations)
    {
      if (first.name == name && rel == Relationship::parent)
      {
        result.push_back(second);
      }
    }
    return result;
  }
};
```

Step 4: Implement Research Class Using Dependency Injection

Instead of directly depending on `Relationships`, `Research` now depends on the `RelationshipBrowser` interface.

```
struct Research // high-level
{
  Research(RelationshipBrowser& browser)
```

```
  {
    for (auto& child : browser.find_all_children_of("John"))
    {
      cout << "John has a child called " << child.name << endl;
    }
  }
};
```

Here, `Research` receives `RelationshipBrowser` as a dependency via its constructor, which is Dependency Injection in action.

Step 5: Inject Dependencies in Main Function

```
int main()
{
  Person parent{"John"};
  Person child1{"Chris"};
  Person child2{"Matt"};

  Relationships relationships;
  relationships.add_parent_and_child(parent, child1);
  relationships.add_parent_and_child(parent, child2);

  Research _(relationships);  // Dependency Injection
  return 0;
}
```

Types of Dependency Injection

1. **Constructor Injection (Used in this example)**: Dependencies are injected via the constructor.
2. **Setter Injection**: Dependencies are assigned through setter methods.
3. **Interface Injection**: A separate interface is used to inject dependencies.

Benefits of Using Dependency Injection

- **Reduces tight coupling** between classes.
- **Easier unit testing** since dependencies can be replaced with mocks.
- **Enhances code flexibility and scalability.**

Conclusion

Dependency Injection enables classes to depend on abstractions rather than concrete implementations. This leads to loosely coupled, easily testable, and maintainable code, making it a key principle in modern software design.

Setter Injection in C++

Introduction to Setter Injection

Setter Injection is a type of Dependency Injection where dependencies are provided via setter methods rather than through the constructor. This allows for more flexibility since dependencies can be changed after object creation.

Why Use Setter Injection?

- **Flexibility:** Dependencies can be changed at runtime.
- **Optional Dependencies:** Objects can work without certain dependencies.
- **Separation of Concerns:** Keeps object initialization and dependency management separate.

Example: Implementing Setter Injection in C++

We will modify our previous `Research` class to receive its dependency through a setter method.

Step 1: Define Relationship and Person Structures

```
enum class Relationship

{

  parent,

  child,

  sibling

};
```

```
struct Person

{

  string name;

};
```

Step 2: Define RelationshipBrowser Interface

This interface abstracts how relationships are accessed.

```
struct RelationshipBrowser

{

  virtual vector<Person> find_all_children_of(const string& name) = 0;

};
```

Step 3: Implement Relationships Class

This class stores relationships and implements `RelationshipBrowser`.

```
struct Relationships : RelationshipBrowser

{

  vector<tuple<Person, Relationship, Person>> relations;



  void add_parent_and_child(const Person& parent, const Person& child)

  {

    relations.push_back({parent, Relationship::parent, child});

    relations.push_back({child, Relationship::child, parent});

  }
```

```
  vector<Person> find_all_children_of(const string &name) override
  {
    vector<Person> result;
    for (auto&& [first, rel, second] : relations)
    {
      if (first.name == name && rel == Relationship::parent)
      {
        result.push_back(second);
      }
    }
    return result;
  }
};
```

Step 4: Implement Research Class Using Setter Injection

Instead of receiving `RelationshipBrowser` via the constructor, we use a setter method.

```
struct Research
{
  RelationshipBrowser* browser = nullptr; // Pointer to
RelationshipBrowser
```

```
  void set_relationship_browser(RelationshipBrowser& b)
  {
    browser = &b;
  }

  void perform_research()
  {
    if (browser) // Ensure dependency is set
    {
      for (auto& child : browser->find_all_children_of("John"))
      {
        cout << "John has a child called " << child.name << endl;
      }
    }
    else
    {
      cout << "No RelationshipBrowser set!" << endl;
    }
  }
};
```

- `set_relationship_browser(RelationshipBrowser& b)` is the setter method.
- `perform_research()` checks if `browser` is set before accessing it.

Step 5: Inject Dependencies in Main Function

```
int main()

{

  Person parent{"John"};

  Person child1{"Chris"};

  Person child2{"Matt"};



  Relationships relationships;

  relationships.add_parent_and_child(parent, child1);

  relationships.add_parent_and_child(parent, child2);



  Research research;

  research.set_relationship_browser(relationships); // Setter Injection

  research.perform_research(); // Perform research after setting
dependency



  return 0;

}
```

Benefits of Setter Injection

- **More flexible** since dependencies can be updated at runtime.
- **Supports optional dependencies**, unlike constructor injection.
- **Better separation of concerns**, allowing dependency management after object creation.

Potential Drawbacks

- **Objects are incomplete** until dependencies are set.
- **Requires additional error handling** to ensure dependencies are provided before usage.

Conclusion

Setter Injection enables dynamic configuration of dependencies. It provides greater flexibility but requires careful handling to ensure objects are properly initialized before use.

Interface Injection in C++

Introduction to Interface Injection

Interface Injection is a form of Dependency Injection where the dependency is injected via a method of an interface that the dependent class implements. Unlike Constructor Injection or Setter Injection, the dependent class does not directly request the dependency; instead, an external entity supplies it by calling a method defined in an interface.

Why Use Interface Injection?

- **Decouples dependency resolution from object creation**
- **Allows dependencies to be dynamically injected**
- **Improves modularity and testability**

Example: Implementing Interface Injection in C++

We will modify our existing example to use Interface Injection.

Step 1: Define Relationship and Person Structures

```
enum class Relationship
{
  parent,
  child,
  sibling
};
```

```
struct Person
{
  string name;
};
```

Step 2: Define RelationshipBrowser Interface

This interface provides a way to access relationships.

```
struct RelationshipBrowser

{

  virtual vector<Person> find_all_children_of(const string& name) = 0;

};
```

Step 3: Implement Relationships Class

This class stores relationships and implements `RelationshipBrowser`.

```
struct Relationships : RelationshipBrowser // low-level

{

  vector<tuple<Person, Relationship, Person>> relations;



  void add_parent_and_child(const Person& parent, const Person& child)

  {

    relations.push_back({parent, Relationship::parent, child});

    relations.push_back({child, Relationship::child, parent});

  }
```

```
  vector<Person> find_all_children_of(const string &name) override

  {

    vector<Person> result;

    for (auto&& [first, rel, second] : relations)

    {

      if (first.name == name && rel == Relationship::parent)

      {

        result.push_back(second);

      }

    }

    return result;

  }

};
```

Step 4: Define an Interface for Dependency Injection

Instead of injecting the dependency through a constructor or setter, we create an interface that provides a method to set the dependency.

```
struct RelationshipConsumer

{

  virtual void set_relationship_browser(RelationshipBrowser& browser) = 0;

};
```

Step 5: Implement Research Class Using Interface Injection

The `Research` class now implements the `RelationshipConsumer` interface.

```
struct Research : RelationshipConsumer

{

  RelationshipBrowser* browser = nullptr;



  void set_relationship_browser(RelationshipBrowser& b) override

  {

    browser = &b;

  }



  void perform_research()

  {

    if (browser)

    {

      for (auto& child : browser->find_all_children_of("John"))

      {

        cout << "John has a child called " << child.name << endl;

      }

    }

    else

    {

      cout << "No RelationshipBrowser set!" << endl;

    }

  }
```

```
};
```

Step 6: Inject Dependencies in Main Function

Instead of injecting the dependency via a constructor or setter, we call the `set_relationship_browser` method from an external entity.

```
int main()

{

  Person parent{"John"};

  Person child1{"Chris"};

  Person child2{"Matt"};



  Relationships relationships;

  relationships.add_parent_and_child(parent, child1);

  relationships.add_parent_and_child(parent, child2);



  Research research;

  research.set_relationship_browser(relationships); // Interface Injection

  research.perform_research();



  return 0;

}
```

Benefits of Interface Injection

- **Dependency resolution is external to the dependent class**
- **Enables dynamic dependency management**
- **Allows multiple implementations of dependencies**

Potential Drawbacks

- **More complexity** compared to Constructor or Setter Injection
- **Requires additional interfaces for dependency injection**

Conclusion

Interface Injection provides a powerful way to decouple dependencies from object creation and management. It ensures that dependencies are set externally, making the system more flexible and maintainable.