Dependency Inversion Principle (DIP) in C++

Introduction to Dependency Inversion Principle

The Dependency Inversion Principle (DIP) is one of the five SOLID principles of object-oriented programming. It primarily focuses on structuring dependencies between objects in a way that improves maintainability and flexibility.

DIP consists of two key ideas:

1. **High-level modules should not depend on low-level modules.** Instead, both should depend on abstractions.
2. **Abstractions should not depend on details.** Details should depend on abstractions.

Understanding Abstraction

An abstraction refers to interfaces or base classes, which provide a generalized way to interact with different implementations. Instead of depending on a concrete class, we depend on an interface that offers the required functionality.

Example of Bad Dependency If a high-level module directly interacts with a low-level module's implementation, any change in the low-level module can break the high-level module. This leads to tight coupling.

Example: Modeling Relationships

Consider a scenario where we model relationships between people using an enum class and structures.

Step 1: Define Relationship and Person Structures

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

struct Person
{
  string name;
};
```

Step 2: Low-Level Module (Relationships Class)

The `Relationships` class is responsible for storing relationships in a vector of tuples.

```
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});
  }
};
```

This class exposes `relations` directly, which can create issues if the storage mechanism changes.

Step 3: High-Level Module (Research Class with Direct Dependency - Bad Design)

```
struct Research // high-level
{
 Research(const Relationships& relationships)
 {
   auto& relations = relationships.relations;
   for (auto&& [first, rel, second] : relations)
   {
     if (first.name == "John" && rel == Relationship::parent)
     {
       cout << "John has a child called " << second.name << endl;
     }
   }
 }
};
```

Issue: The `Research` class directly accesses `relationships.relations`. If `relations` becomes private or changes to a different data structure, the `Research` class breaks.

Applying Dependency Inversion Principle

To fix this, we introduce an abstraction (`RelationshipBrowser`) that provides an interface for finding relationships.

Step 4: Introduce RelationshipBrowser Interface

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

Now, instead of directly exposing `relations`, `Relationships` will implement `RelationshipBrowser`.

Step 5: Modify Relationships Class

```
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 6: Modify Research Class to Use Abstraction

```
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;
    }
  }
};
```

Now, `Research` depends only on the abstraction (`RelationshipBrowser`), making it independent of the actual data storage implementation.

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

Final Code and Execution

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);
/*
  The underscore (_) used in Research _(relationships); is just a variable
name.In C++, every object must have a name unless it's an anonymous
temporary object.
*/

  return 0;
}
```

Benefits of Applying Dependency Inversion Principle

- **Loose Coupling:** `Research` does not depend on `Relationships`, but on an abstraction (`RelationshipBrowser`).
- **Flexibility:** We can change the internal storage in `Relationships` without affecting `Research`.
- **Testability:** We can create mock implementations of `RelationshipBrowser` for unit testing `Research`.

Conclusion

The Dependency Inversion Principle ensures that high-level modules are not tightly coupled to low-level modules. Instead, both depend on abstractions, improving maintainability and scalability of the code.