Enterprise Application Systems (EAS)

Dependency Injection

Daniel Fink

Institute for Computational Physics

February 01, 2023

What again was EAS?

Institute of Architecture of Application Systems

An (enterprise) application system is a software that directly supports one or more business activities of a company. Typical attributes for such a system are:

- It covers essential parts of the business activities of a company.
- It processes large amounts of persistent data.
- It allows simultaneous use by a large number of users.
- It exchanges information with similar systems.

What is the Problem?

- We use many different services to process information.
- The services are coupled with each other.
- Coupling introduces complexity, which
  - is hard to maintain.
  - makes it difficult to change the system.
  - is not scalable.
  - decreases reliability.

What is the Problem?

- We use many different services to process information.
- The services are coupled with each other.
- Coupling introduces complexity, which
  - is hard to maintain.
  - makes it difficult to change the system.
  - is not scalable.
  - decreases reliability.
- Coupling is unavoidable.
- However, we want to achieve a *loose coupling*.
- The *Inversion of Control*(IoC) principle should be used.

The Inversion of Control (IoC) Principle

ChatGPT

The IoC principle refers to a design approach where the flow of control in a software application is reversed. Instead of the code determining what objects it needs and creating them, the objects are created outside of the code and provided to it when needed.

The Inversion of Control (IoC) Principle

ChatGPT

The IoC principle refers to a design approach where the flow of control in a software application is reversed. Instead of the code determining what objects it needs and creating them, the objects are created outside of the code and provided to it when needed.

- Dependency Injection (DI) is one way to achieve IoC.
- Alternatives would be:
  - Factory Method Pattern
  - Abstract Factory Pattern
  - Service Locator Pattern
  - Mediator Pattern
  - ...

The Dependency Injection (DI) Design Pattern

ChatGPT

Dependency Injection is a design pattern that implements the Inversion of Control (IoC) principle. It allows a class to receive its dependencies from an external source rather than creating them itself. This makes the class more flexible and loosely coupled, as the dependencies can be easily changed without affecting the class.

The Dependency Injection (DI) Design Pattern

ChatGPT

Dependency Injection is a design pattern that implements the Inversion of Control (IoC) principle. It allows a class to receive its dependencies from an external source rather than creating them itself. This makes the class more flexible and loosely coupled, as the dependencies can be easily changed without affecting the class.

Loose Coupling with Classes - Python

```python
import os

class ApiClient:
    def __init__(self):
        self.api_key = os.getenv("API_KEY")  # <-- dependency
        self.timeout = int(os.getenv("TIMEOUT"))  # <-- dependency
```

Loose Coupling with Classes - Python

```python
import os

class ApiClient:
    def __init__(self):
        self.api_key = os.getenv("API_KEY")  # <-- dependency
        self.timeout = int(os.getenv("TIMEOUT"))  # <-- dependency

class Service:
    def __init__(self):
        self.api_client = ApiClient()  # <-- tight coupling
```

Loose Coupling with Classes - Python

```python
import os

class ApiClient:
    def __init__(self):
        self.api_key = os.getenv("API_KEY")  # <-- dependency
        self.timeout = int(os.getenv("TIMEOUT"))  # <-- dependency

class Service:
    def __init__(self):
        self.api_client = ApiClient()  # <-- tight coupling
```

- What if we want to store the API key in a file?
- What if we want the timeout to be determined by another service?
- The *ApiClient* should not need to know these details.

Loose Coupling with Classes - Python

```python
import os

class ApiClient:
    def __init__(self, api_key, timeout):
        self.api_key = api_key  # <-- dependency is injected
        self.timeout = timeout  # <-- dependency is injected

class Service:
    def __init__(self):
        api_key = os.getenv("API_KEY")
        timeout = int(os.getenv("TIMEOUT"))
        self.api_client = ApiClient(api_key, timeout)  # <-- tight coupling
```

Loose Coupling with Classes - Python

```python
import os

class ApiClient:
    def __init__(self, api_key, timeout):
        self.api_key = api_key  # <-- dependency is injected
        self.timeout = timeout  # <-- dependency is injected

class Service:
    def __init__(self):
        api_key = os.getenv("API_KEY")
        timeout = int(os.getenv("TIMEOUT"))
        self.api_client = ApiClient(api_key, timeout)  # <-- tight coupling
```

- The *ApiClient* is fine now.
- However, we still have a tight coupling between the *Service* and the *ApiClient*.

Loose Coupling with Classes - Python

```python
class ApiClient:
    def __init__(self, api_key, timeout):
        self.api_key = api_key  # <-- dependency is injected
        self.timeout = timeout  # <-- dependency is injected

class Service:
    def __init__(self, api_client):
        self.api_client = api_client  # <-- loose coupling
```

- Now, we have a loose coupling, since neither the *ApiClient* nor the *Service* needs to know any implementation details of the other one.

Loose Coupling with Classes - Python

```python
class ApiClient:
    def __init__(self, api_key, timeout):
        self.api_key = api_key  # <-- dependency is injected
        self.timeout = timeout  # <-- dependency is injected

class Service:
    def __init__(self, api_client):
        self.api_client = api_client  # <-- loose coupling
```

- Now, we have a loose coupling, since neither the *ApiClient* nor the *Service* needs to know any implementation details of the other one.
- However, with static types, this is not possible.

Loose Coupling with Classes - C#

```
public class ApiClient {
    private readonly string _apiKey;
    private readonly int _timeout;

    public ApiClient(string apiKey, int timeout) {
        _apiKey = apiKey;
        _timeout = timeout;
    }
}
```

Loose Coupling with Classes - C#

```
public class ApiClient {
    private readonly string _apiKey;
    private readonly int _timeout;

    public ApiClient(string apiKey, int timeout) {
        _apiKey = apiKey;
        _timeout = timeout;
    }
}
public class Service {
    private readonly ApiClient _client;

    // This class depends on the type "ApiClient",
    // which is an implementation detail.
    public Service(ApiClient client) {
        _client = client;
    }
}
```

Loose Coupling with Classes - C#

```
public interface IApiClient {
    // Only publicly accessible methods and properties, e.g.,
    public string GetData();
}
```

Loose Coupling with Classes - C#

```
public interface IApiClient {
    // Only publicly accessible methods and properties, e.g.,
    public string GetData();
}

public class ApiClient : IApiClient {
    private readonly string _apiKey;
    private readonly int _timeout;

    public ApiClient(string apiKey, int timeout) {
        _apiKey = apiKey;
        _timeout = timeout;
    }

    public string GetData() {
        // ...
    }
}
```

Loose Coupling with Classes - C#

```
public class Service {
    private readonly IApiClient _client;

    // This class now depends on the type "IApiClient",
    // which is only an interface, not a real implementation.
    public Service(IApiClient client) {
        _client = client;
    }
}
```

Loose Coupling with Classes - C#

```
// Create the "DI Container"
ServiceCollection collection = new ServiceCollection();

// Register the intend to get an "IApiClient" and
// associate it with the implementation "ApiClient"
collection.AddTransient<IApiClient, ApiClient>();
collection.AddTransient<Service>();

// Build the "Service Provider"
ServiceProvider provider = collection.BuildServiceProvider();

// Get the "Service"
Service service = provider.GetService<Service>();
```

Advantages

- A change of *ApiClient* does not change *Service*.
- *Service* can be unit-tested without *ApiClient*:

```
// Register a dummy client instead of a real client
collection.AddTransient<IApiClient, ApiClientDummy>();
```

Advantages

- A change of *ApiClient* does not change *Service*.
- *Service* can be unit-tested without *ApiClient*:

```
// Register a dummy client instead of a real client
collection.AddTransient<IApiClient, ApiClientDummy>();
```

- One can easily exchange layers of the application, e.g.,

```
// Register a MySQL database
collection.AddTransient<IDatabase, MySqlDatabase>();
// A few months later ...
// Register an in-memory database
collection.AddTransient<IDatabase, InMemoryDatabase>();
```

Advantages

- One can easily control the lifetime of the services, e.g.,

```
// Transient = every time a new instance
collection.AddTransient<IApiClient, ApiClientDummy>();
// Singleton = every time the same instance
collection.AddSingleton<IDatabase, MySqlDatabase>();
```

Advantages

- One can easily control the lifetime of the services, e.g.,

```
// Transient = every time a new instance
collection.AddTransient<IApiClient, ApiClientDummy>();
// Singleton = every time the same instance
collection.AddSingleton<IDatabase, MySqlDatabase>();
```

- Better separation of concerns (cf. Single Responsibility principle).
- Improved reusability: services can be used in different contexts.

Disadvantages

- The startup procedure typically explodes.
- Code might be difficult to trace and read because the behavior is separated from the construction.
- It shifts some compile-time errors to run-time errors (forgot to register a type).

Outlook

- Dipendency Injection is heavily used for middlewares.
- It allows an elegant implementation of the Mediator pattern.
- It is very useful to perform *serverless (cloud) computing*.