08 - Frameworks and Inversion of Control

Software Architect

Organizes the architecture of the system to be developed and the architecture of how the work should be organized. It must be aware of all the technologies employed in the project.

Frameworks

A framework consist of parts that are found in many apps of that type:

- Libraries;
- Engines: ready-made extensible programs;
- Tools: optional, useful for development, configuration...;

We can have two type of frameworks:

- Software Framework: a collection of common code providing generic functionality that can be selectively overridden or specialized by user code providing specific functionality;
- Application Framework: a software framework used to implement the standard structure of an application for a specific development environment.

We can extend a framework to meet the needs of a particular application:

- Within the framework language: sub-classing, implementing interfaces, registering event handlers...;
- Plug-ins: the framework loads extra code in a specific format.

Comparing frameworks to IDEs, we can observe that they are orthogonal concepts:

- A framework can be supported by several IDEs (e.g. Spring supported by NetBeans);
- An IDE can support several frameworks (NetBeans support Spring).

Inversion of Control

Frameworks, like libraries, can provide reusable abstractions (e.g. abstract data structure) that the user must implement in order to make use of the full framework.

However, the overall program's flow of control is not dictated by the caller, but by the framework: this is called inversion of control. The framework determines the application architecture: the user generally implements a few callback functions or specializes a few classes, and then invokes a single method or procedure. This opposes to what a library does, where the user calls code has the control over the architecture and the library contains the implementation.

Dependency Injection

Often, inversion of control also concerns dependencies, coupling and configuration.

Dependency injection is an instance of Inversion of Control, where dependencies are passed into an object through constructors/setters/service lookups, which the object will depend on in order to behave correctly. It is a way to provide decoupling between two objects, as it helps improving extensibility (e.g. changing a database with another one), testability (we're testing an object which is using another object) and reusability.

There are 3 types:

- Setter Injection: the setter methods are called on the user's beans after invoking a no-argument constructor or no-argument static factory method to instantiate their bean. It leverages existing JavaBean reflective patterns, but it makes possible to create partially constructed objects and dependencies can be changed over time (can be either a bug or a feature);

```language-java
public class TradeMonitor{
        private LimitRepository limitRepo;

        // NOTE: we do not instantiate the field limitRepo
        public TradeMonitor(){}

        public void setLimitRepository(LimitRepository limitRepo){
                this.limitRepo = limitRepo;
        }

        // NOTE: we must check that limitRepo != null
        public bool TryTrade(String tradeOp, int amount) { ... }

}
```

- Constructor Injection: a constructor is invoked with a number of arguments, each representing a collaborator. Using this we can validate that the injected beans are not null and fail at compile time. Differently from Setter Injection, objects can't be partially constructed, but class evolution could grow complicated, as constructors could get big and parameters become confusing;
- Interface Injection: never used.

Opposed to Dependency Injection as another solution to solve decoupling, we could use a ServiceLocator, which is a static registry of the components you need. The registry saves the name and/or the type of the component and when a component needs another component it queries the registry (using name and/or type). ServiceLocator allows new components to be dynamically created and used by other components later. However, every component needs to be registered to the service locator:

- If bound by name: service can't be type-checked (as we only get the name) and an object depends on the dependent component names (what if dependent component changes name? We would need to modify the name looked in the registry in every code);
- If bound by type: we can have only two objects of the same type, allowing one instance per type in the container;

While with Dependency Injection there is no explicit request (as components appears in the application class) and it is easier to find dependencies of component (check constructors and setters vs check all invocations to locator in the source code), it might be harder to understand than the Service Locator, which in turns have the application still dependent to the locator.

Spring

Spring is a framework used to develop java client-server web app. It heavily employs dependency injection and it is very light.

The objects that form the backbone of a Spring application are called beans: a bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

An Inversion of Control container have configuration info about the objects that can be instantiated and assure that the dependencies between objects are satisfied. Most IoC containers support auto-wiring: automatic wiring between properties of a bean and other beans based (e.g. name or type). In particular, bean's definition contains the information called configuration meta-data, which is needed for the container to know the following:

- How to create a bean;
- Bean's lifecycle details;
- Bean's dependencies.

Framework Construction

A software family is a set of different solutions for a common problem. An aspect of a solution can be implemented by one or more methods.

In a software family we can have some:

- Frozen Spot: *common* aspect of the family. The implementation is generally done through one or more Template Method, a concrete method of an abstract class;
- Hot Spot: *variable* aspect of the family. The implementation can't be provided by the Framework, so the framework exposes one or more Hook Method, which is an abstract method that must be implemented by the user. Indeed, to completely implement an hot spot we will need to realize an Hot Spot Subsystem: an abstract base class + some concrete subclasses.

There are 2 principles than can be exploited in order to implement a Hot Spot Subsystem:

- Unification (Template Method Design Pattern): it is a bottom-up approach. It uses inheritance for the implementation: an abstract base class defines template/hook methods and leave to sub-classes the implementation of the hook methods. The algorithm selection happens at compile-time;
  - In details: the Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single template method. The steps may either be `abstract`, or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself);
- Separation (Strategy Design Pattern): it is a top-down approach. It uses delegation (composition) for the implementation, directly implementing the template methods in a concrete context class and defining the hook methods in a separate abstract class. Then, we will have subclasses of the abstract class that will implement the hook methods, which will also make use of the delegates of the template methods. Thus, template methods are separated from hook methods. The algorithm selection happens at run-time;
  - In details: the Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.
    The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.
    The context isn’t responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn’t know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.
    This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.

Template Method works at the class level, so it’s static. The client and the chosen algorithm are tightly coupled. *Strategy* works on the object level, letting you switch behaviours at runtime. Thanks to dependency injection, it is not tightly coupled. The two patterns could easily be used together. You might have a strategy pattern where several implementations belong to a family of strategies implemented using a template pattern.

In conclusion, the steps to design a framework are:

- Identify software family;
- Identify frozen spots and hot spots;
- Exploit design patterns and other techniques to achieve as generality as possible and to reduce coupling to the possible minimum. Inversion of Control and dependency injection arises naturally and are keys to framework design.