Advanced Programming

Software is everywhere (AI, Big data, data analysis etc) so in order to have the chance to get satisfactory job in their life to develop interesting system is needed to learn how to program properly. Programming in this century is much more complex than before in terms of scale of programming but the logic behind it is the same as before. Data comes to multiple from multiple sources (DB for Example), and the application need to be distributed on more than one computer and are programmed by group of people.

In order to manage the complexity of this way of programming we need tools which are innovative, original, which resolve our problem:

* **Object oriented programming** was extremely successful in order to give the programmer a tool to build very complex application (e.g., graphical interfaces) but, again, they provide limited support for reuse, OS+ libraries are not enough to support their use ence forth reusable components are needed.

Key ingredients for complex software:

* **Advanced features:** extending programming languages.
* **Component models:** to ensure reusability.
* **Frameworks:** to support efficient development of (component based) applications.
* **Execution environments:** providing runtime support for ever dynamic software systems.

The software architect is a prominent role in any team developing software, his role is not to write the program itself but to organize the layout of the process which will develop in the writing of code in the most efficient way (?).

* **Software frameworks:** 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.

Frameworks, like software libraries, provide reusable abstractions of code wrapped in a well-defined API.

But in **Inversion of Control**, unlike in libraries, the overall program’s flow of control is not dictated by caller, but by framework. Helps solving recurring design problems such as:

* **Drives solution** provides a default behaviour and dictates how to fill-in-the-blanks. It provies you an architecture of program that is already tested.
* **Non-modifiable framework code** gives extensibility (usually by selective overriding).

It is important to understand how a framework Is developed, this is an intellectual challenge which requires a deep understanding of the problem domain and requires mastering of software (design) patterns, OO methods and polymorphism.

**Design Patterns**

Design patterns are fundamental concept in SE and programming and they can be considered as a pre-coocked solution to problem. They are made so a developer facing any given problem can use one of these and possibly he can also adapt to the specifical situation. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Patterns can be applied in more than one area of human knowledge but are very prominent in SI. From here we define:

* **Software design patterns:** are a general, reusable solution to a commonly occurring problem within a given context in software design.

There are different abstraction levels:

* Complex design fo an entire application or subsystem which is a pattern (e.g. model view controller for web application)
* Solution to a general design problem in a particular context (e.g. deciding how entities interact in distributed or concurrent systems)
* Simple reusable design call such as  **linked list, hash tables, etc..**

The pattern provide a solution for software structural problems like:

* Abstraction
* Encapsulation
* Information hidig
* Separation of concerns
* Coupling and cohesion
* Separation of interface and implementation
* Single point of reference
* Divide and conquer

Besides the software structural problems, patterns solve also non-fuctional problems:

* Changeability
* Interoperability
* Efficiency
* Reliability
* Testability
* Reusability

The main componens of a design patterns are:

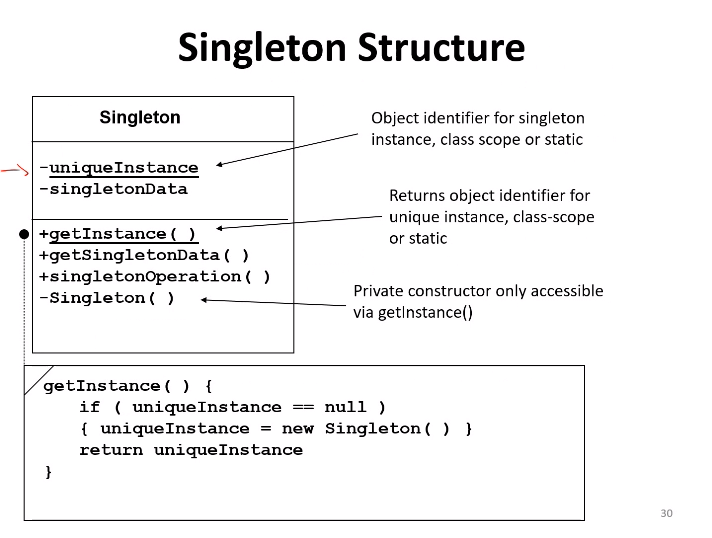
* **Name:** meaniful text that reflects the problem. The name should immediately remind you of the design pattern
* **Problem addressed:** intent of the pattern, objectuves achieved within certain constraints
* **Context:** circumstances under which it can occur, used to determine applicability
* **Forces:** describe the constrains or issues that solution must address, forces may conflict!
* **Solution:** is the static and dynamic relationships among the pattern components. Structure, participants, collaboration. Solution must resolve all forces

Singleton:

Is a creational pattern: is a solution you can adopt while you are creating an object. It answers to the problem of how can we guarantee that one and only one instance of a class can be created? In some application it is important to have exactly one instance of a class, e.g. sales of one company.

**Forces:** can make an object globally accessible as a global variable, but this violates incapsulation. Could use calss (static) operation and attributes, but polymorphic redefinition is not always possible

**Solution:** create a class with class operation getInstance(). When the class is first accessed, this creates relevant object instance and returns object identity to client. On subsequent calls of getInstance(), no new instance is created, but identity of existing object is returned.



To specify a class has only one instance, we make it inherit from Singleton and that allows us to:

* Get controlled access to single object instance through singleton encapsulation
* Tailor for any finite number of instances
* Allow the namespace to not be extended by global variables:
  + Access requires additional message passing because you need getInstance instead getting directly what you need from the constructor.
  + Pattern limits flexibility, significant redesign if singleton class later gets many instances.

Design pattern vs Framework

* More abstract than frameworks:
  + Frameworks can be emboidied in code, but only examples of patterns can be emboidied in code
  + Design patterns explain the intent, trade-offs, and the consequence of a design
* Smaller architectural elements than framevworks:
  + A typical framework contains several design patterns but the reverse is never true.
* Less specialized than frameworks:
  + Frameworks always have a particular application domain
  + Design patterns can be used in nearly any kind of application