301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

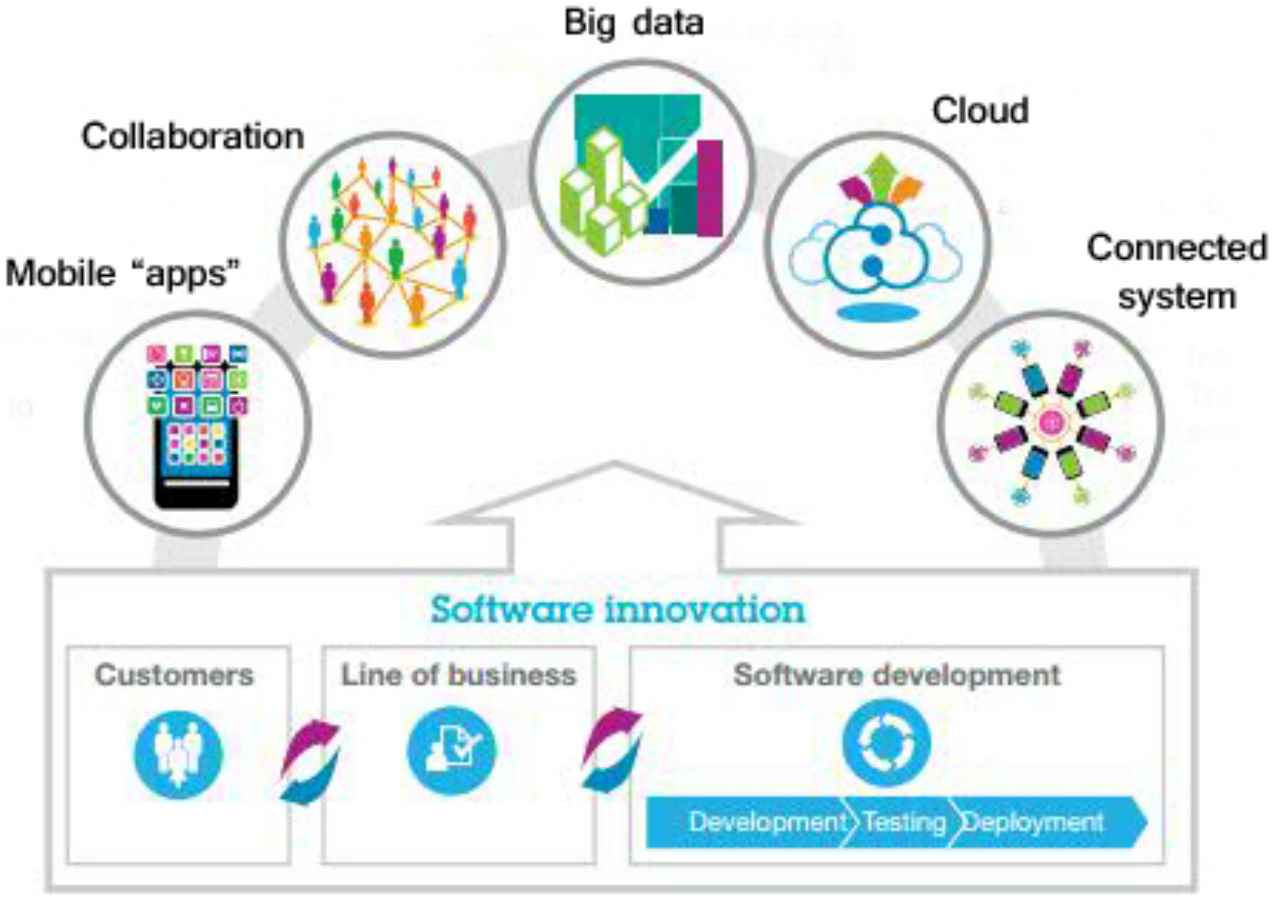
http://pages.di.unipi.it/corradini/

Course pages:

http://pages.di.unipi.it/corradini/Didattica/AP-19/

AP-02*: Motivations and Introduction*

Software is Everywhere



Programming in the 21 century

- Software as complex as ever
- Command line interface not enough
- Data comes from multiple sources: structured (DB) and unstructured
- Single computer not enough
- Software development is a group activity
- Deployment on Web or mobile devices

Complexity Prompts for Innovation

- Object-Oriented Programming allows ever larger applications to be built
- But limited support for reuse
- OS + libraries not enough
- Reusable components are needed
- Multi-tier applications development increases the choices on how to build applications

Key Ingredients for Complex Software

- **Advanced features** extending programming languages

SUCH AS POLIMORPHISM, FOR EXAMPLE

- **Component models** to ensure reusability
- **Frameworks** to support efficient development of (component based) applications
- **Execution environments** providing runtime support for ever dynamic software systems

A BIT MORE LOW LEVEL THAN THE OTHERS

THEY GIVE US A SKELETON, SO WE DON'T WRITE EVERYTIME FROM SCRATCH

THIS 4 ALLOWS US TO "ATTACK" COMPLEXITY

The Software Architect

- A new role is needed: **Software Architect**
- to create, define or choose an **application framework**
- to create the component design according to a **component model**
- to structure a complex application into pieces
- to understand the interactions and dependencies among components
- to select the **execution environment / platform** based on cost/performance criteria
- to organize and supervise the development process

Course Objectives

- Understand programming language technology:
  - Execution Models
  - Run-time systems
- Analyze programming metaphors:
  - Objects
  - Components
  - Patterns
- Learn advanced programming techniques
- Present state-of-the-art frameworks incorporating these techniques
- Practice with all these concepts through small projects

Course Syllabus

- Programming Languages and Abstract Machines
- Run Time Systems and the JVM
- Component-based Programming
- Software Frameworks and Inversion of Control
- Polymorphism and Generic Programming
- Functional programming and Haskell
- Scripting Languages and Python

Programming Languages and Abstract Machines

- Syntax, Semantics and Pragmatics of PLs
- Programming languages and Abstract Machines
- Interpretation vs. Compilation vs. Mixed
- Examples of Virtual Machines
- Examples of Compilation Schemes

Run-Time Systems and the JVM

- RTSs provide a Virtual Execution Environment interfacing a program in execution with the OS.
- They support, among others:
  - Memory Management, Thread Management
  - Exception Handling and Security
  - AOT and JIT Compilation
  - Dynamic Link/Load
  - Debugging Support and Reflection
  - Verification
- A concrete example: the Java Virtual Machine

Component-based Programming

- Component models and frameworks, an Introduction
- Examples of component-based frameworks:
  - JavaBeans and NetBeans
  - Spring and Spring Beans
  - COM
  - CLR and .NET
  - OSGi and Eclipse
  - Hadoop Map/Reduce

Software Frameworks and Inversion of Control

- **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

Framework Features

- Frameworks, like ***software libraries***, provide ***reusable abstractions*** of code wrapped in a well-defined API
- But: **Inversion of control**
  - unlike in libraries, the overall program's flow of control is not dictated by the caller, but by the framework
- Helps solving recurring design problems
- Drives solution
  - Provides a default behavior
  - Dictates how to fill-in-the-blanks
- Non-modifiable framework code
  - Extensibility: usually by selective overriding

SOFTWARE FRAMEWORK (WHICH CONTAINS THE MAIN) CALLS THE PROGRAM WRITTEN BY THE USER

IT IS THE OPPOSITE OF CLASSIC LIBRARIES: THERE IS AN INVERSION OF CONTROL, THE PROGRAM FLOW IS HANDLED BY THE FRAMEWORK

Examples of Frameworks

- General software frameworks
  - **.NET, Android SDK, Spring, Cocoa, Eclipse, …**
- Frameworks for Application with GUI
  - **MFC, Gnome, Qt, …**
- Web Application Frameworks [based on Model-View-Controller design pattern]
  - **ASP.NET, GWT, Rails, …**
- Concurrency
  - **Hadoop Map/Reduce**

Framework Design

- Intellectual Challenging Task
- Requires a deep understanding of the problem domain
- Requires mastering of **software (design) patterns**, OO methods and **polymorphism** in particular

Polymorphism and Generic Programming

- A classification of Polymorphism
- Polymorphism in C++: inclusion polymorphism and templates
- Java Generics
- The Standard Template Library: an overview
- Generics and inheritance: invariance, covariance and contravariance

Functional programming and Haskell

- Introduction to Functional Programming
- Evaluation strategies (lambda-calculus)
- Haskell: main features
- Type Classes and overloading
- Monads
- Functional programming in Java
  - Lambdas and Stream API

Scripting Languages and Python

- Overview of scripting languages
- Main features of Python
- Imperative, functional and OO programming in Python
- Higher-order functions and Decorators
- On the implementation of Python: the Global Interpreter Lock

Selected Advanced Concepts in Programming Languages

- The RUST programming language
  - Avoiding Aliases + Mutable: Ownership and borrowing
  - Traits, generics and inheritance
- Closures vs Delegates in CLI
- Active patterns in F#
- Extensions in Swift

Design Patterns

Design Patterns in few slides

- A fundamental concept in Software Engineering & Programming, useful whenever one is designing a solution to a problem
- We shall meet several Design Patterns along the course (e.g., *Observer* or *Publish-Subscribe, Visitor, Template Method,*…)
- Just a brief introduction…

Design Patterns: From Architecture to Software Development

- Invented in the 1970's by architect Christopher Alexander:

"*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*"

Christopher Alexander, A Pattern Language, 1977

- The book includes 253 patterns for architectural design
- Common definition of a pattern:

"A solution to a problem in a context."

- Patterns can be applied to many different areas of human endehavour, including software development (where they are more successful!)

THEY HELP TO REUSE CODE!

(Software) Design Patterns

- A **(software) design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design.
- Different abstraction levels:
  - Complex design for an entire application or subsystem
  - Solution to a general design problem in a particular context
  - Simple reusable design class such as ***linked list***, ***hash table***, etc.

Patterns solve **software structural problems** like:

PROBLEMS ABOUT A SPECIFIC FEATURE OF OUR APPLICATION: A QUANTITATIVE POV

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

Patterns also solve **non-functional problems** like:

NON-FUNCTIONAL PROPERTIES THAT WE WANT TO HAVE IN OUR APPLICATION: QUALITATIVE POV

- Changeability
- Interoperability
- Efficiency
- Reliability
- Testability
- Reusability

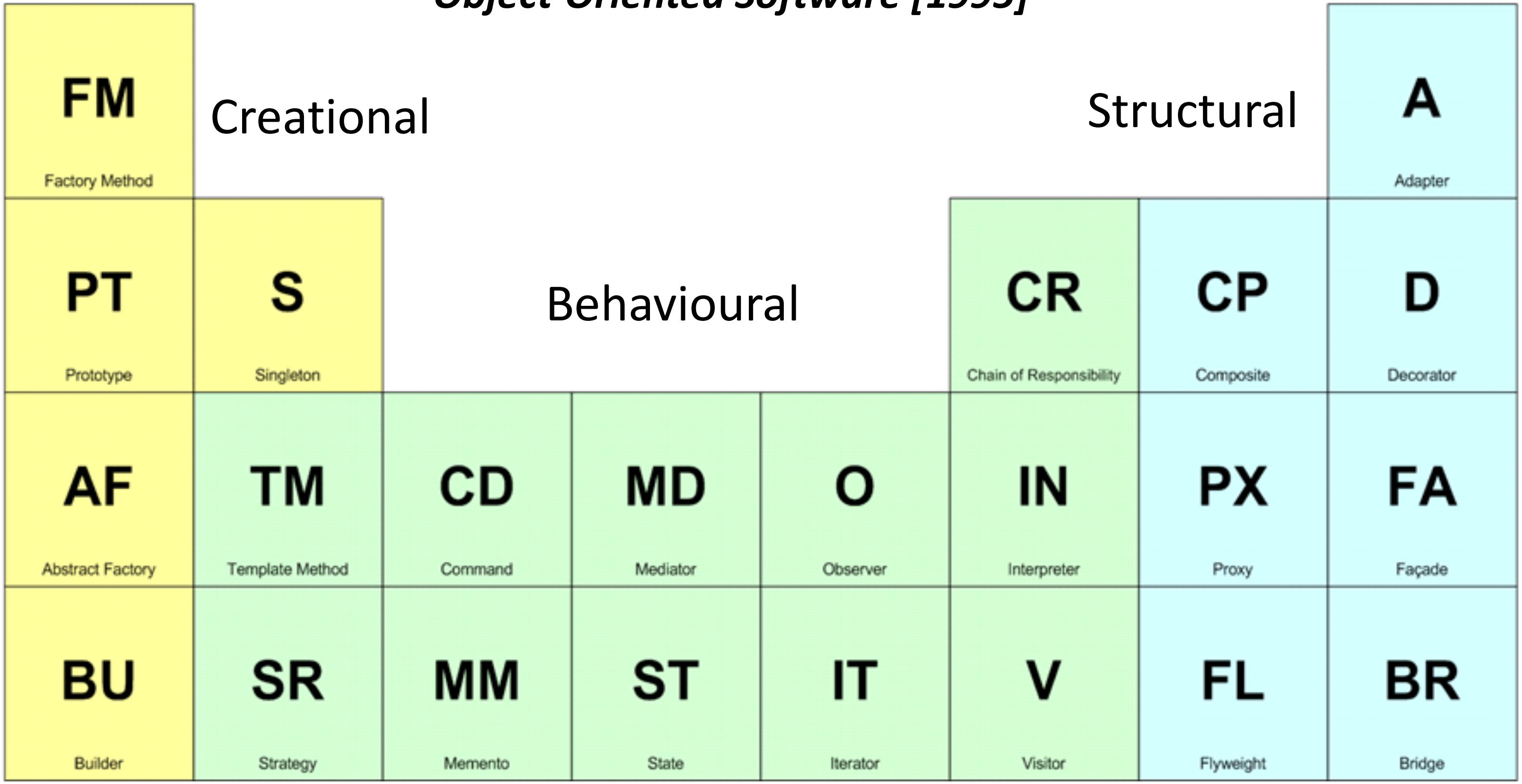
Main components of a Design Pattern

- **Name**: meaningful text that reflects the problem, e.g. Bridge, Mediator, Flyweight
- **Problem addressed**: intent of the pattern, objectives achieved within certain constraints
- **Context**: circumstances under which it can occur, used to determine applicability
- **Forces**: constraints or issues that solution must address, forces may conflict! (IMPOSED BY THE CONTEXT)
- **Solution**: the static and dynamic relationships among the pattern components. Structure, participants, collaboration. Solution must resolve all forces!

The 23 Design Patterns of the Gang of Four

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

Design Patterns: Elements of Reusable Object-Oriented Software [1995]



Creational

Structural

Behavioural

FM Factory Method							A Adapter
PT Prototype	S Singleton				CR Chain of Responsibility	CP Composite	D Decorator
AF Abstract Factory	TM Template Method	CD Command	MD Mediator	O Observer	IN Interpreter	PX Proxy	FA Façade
BU Builder	SR Strategy	MM Memento	ST State	IT Iterator	V Visitor	FL Flyweight	BR Bridge

5.5. Pattern: Singleton (Creational)

CREATIONAL CATEGORY : SET OF DP USED TO CREATE SOME OBJECTS

Name: Singleton

Problem:

How can we guarantee that one and only one instance of a class can be created?

Context: In some applications it is important to have exactly one instance of a class, e.g. sales of one company.

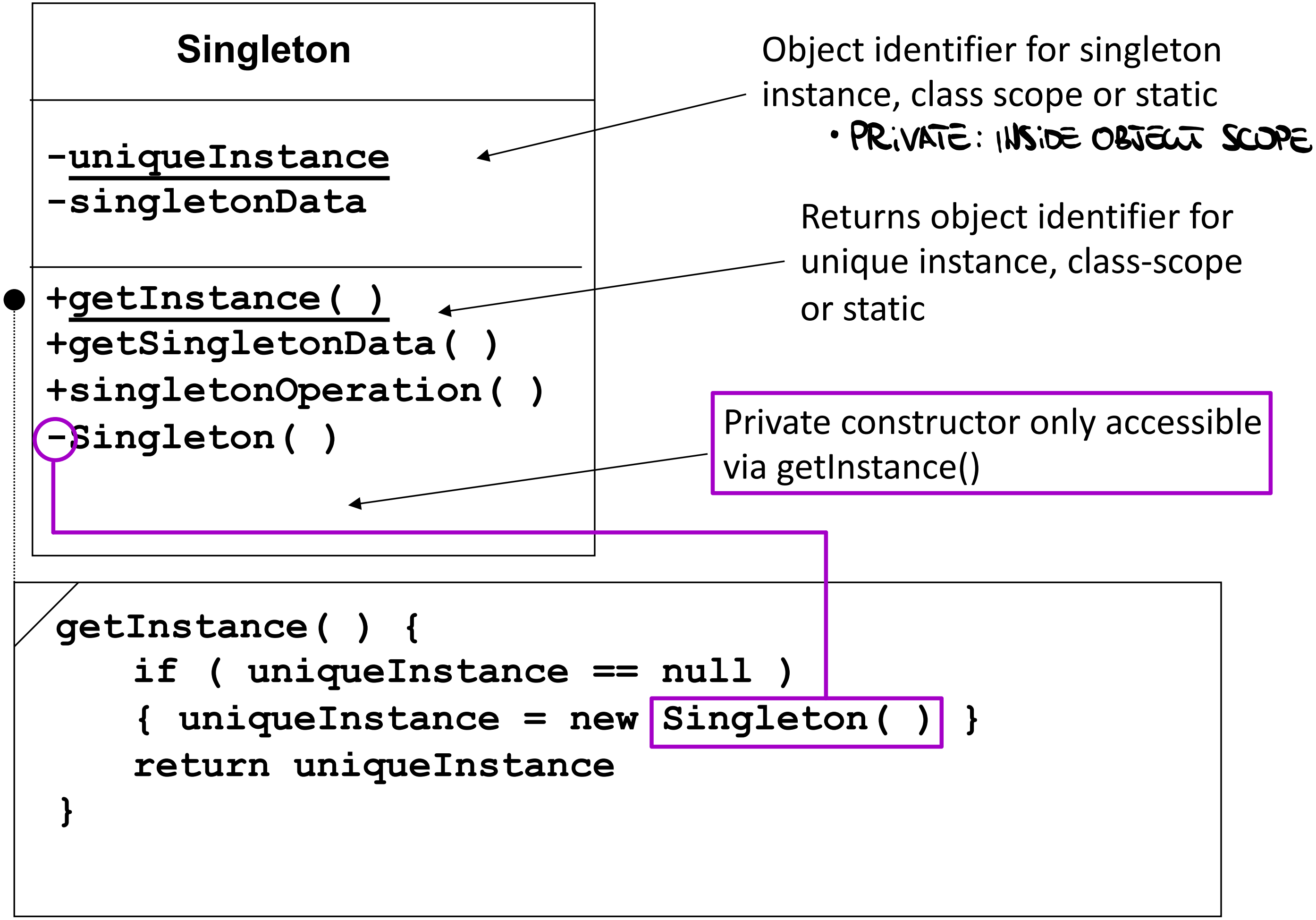
Forces: Can make an object globally accessible as a global variable, but this violates encapsulation. Could use class (static) operations and attributes, but polymorphic redefinition is not always possible.

HAS TO BE ALWAYS POSSIBLE

Solution:

Create a class with a class operation **getInstance()**. When 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.

Singleton Structure



Singleton

```
-uniqueInstance
-singletonData

+getInstance( )
+getSingletonData( )
+singletonOperation( )
-Singleton( )
```

Object identifier for singleton instance, class scope or static

- PRIVATE: INSIDE OBJECT SCOPE

Returns object identifier for unique instance, class-scope or static

Private constructor only accessible via getInstance()

```
getInstance( ) {
    if ( uniqueInstance == null )
    { uniqueInstance = new Singleton( ) }
    return uniqueInstance
}
```

Example: Code

```
class Singleton {

  private static Singleton uniqueInstance = null;

  private Singleton( ) { .. } // private constructor

  public static Singleton getInstance( ) {

   if (uniqueInstance == null)

       uniqueInstance = new Singleton(); //call constructor

   return uniqueInstance;

  }

}
```

Comments

- To specify a class has only one instance, we make it inherit from **`Singleton`**.

+ controlled access to single object instance through **`Singleton`** encapsulation

+ Can tailor for any finite number of instances

+ namespace not extended by global variables

- access requires additional message passing

- Pattern limits flexibility, significant redesign if singleton class later gets many instances

IF AT SOME POINT I DON'T NEED SINGLETON ANYMORE WE WILL HAVE TO MODIFY A LOT OF CODE!

Design Patterns vs. Frameworks

- *More abstract than frameworks*

  DP ARE CONCEPTUAL
  FW ARE PRACTICAL

  - Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code
  - Design patterns explain the intent, trade-offs, and consequences of a design
- *Smaller architectural elements than frameworks*
  - 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