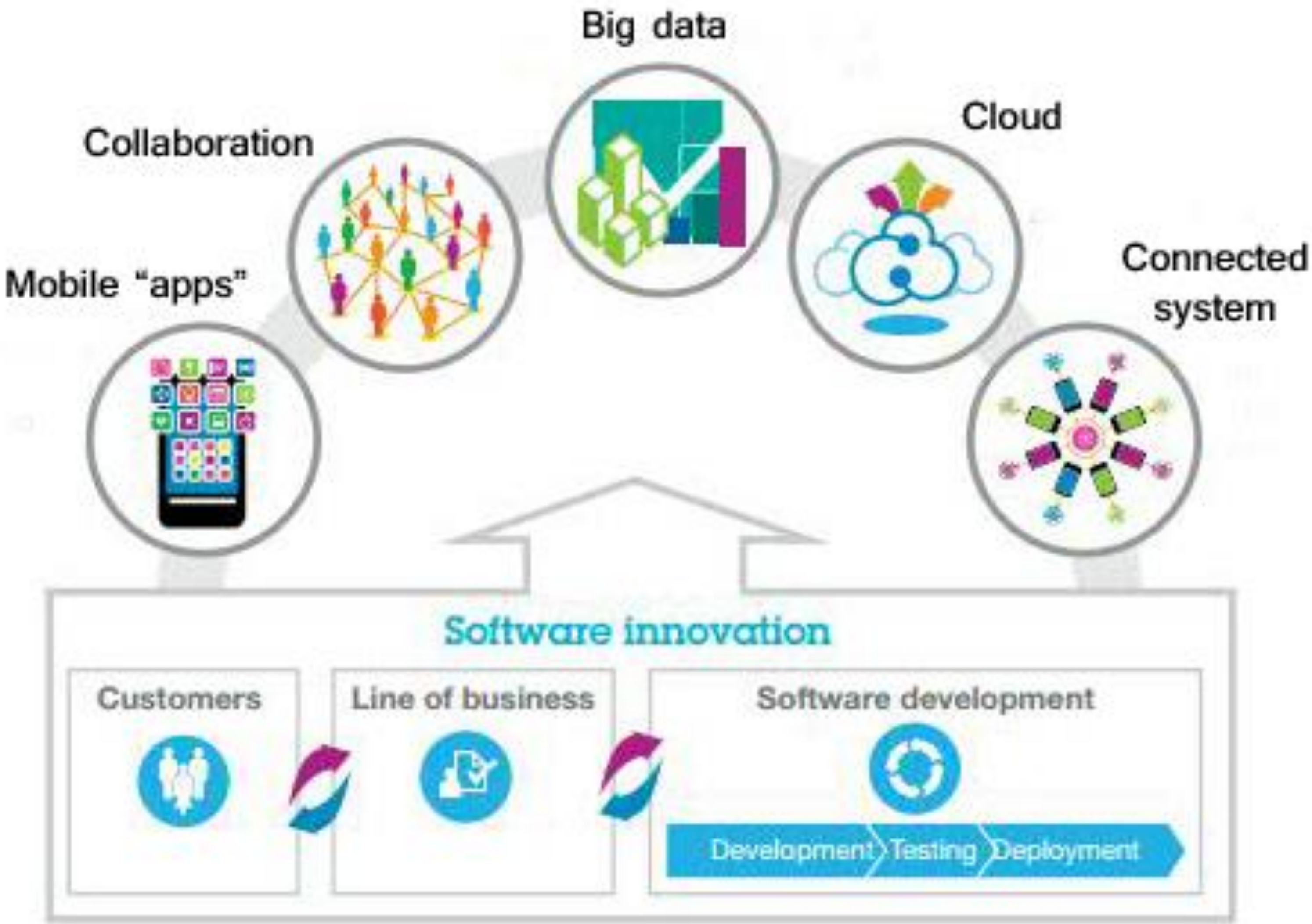
301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea.corradini@unipi.it
http://pages.di.unipi.it/corradini/

Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-24/

AP-02*: Motivations and Introduction*

Software is Everywhere



Software is Everywhere (by ChatGPT)

Programming in the 21 century

Increased Software Complexity:

- Diverse Platforms and Environments
- Multiple Data Sources: Structured (DB) and Unstructured (logs, social media feeds)
- Concurrency and Scalability
- Security and Compliance
- Collaboration and Group Development
- Rapid Technological Change
- Deployment and Maintenence

Complexity Prompts for Innovation

- Software Complexity requires innovative ways to Manage, Simplify, Accelerate the Development Process
- **Software Reuse** is needed to reduce Development Time, Lower Costs, and Improve Reliability
- **Object-Oriented Programming** allows ever larger applications to be built, but limited support for reuse:
  - Reuse granularity: only at the level of classes
  - Inheritance induces tight coupling
  - Difficulty in Cross-Project Reuse
  - Interoperability issues for heterogeneous or multi-language environments

Some Key Ingredients for Complex Software

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

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 Language Pragmatics
- Run Time Support and Execution Environments: the **Java Virtual Machine**
- Components based programming and Frameworks
- **Polymorphism**: a classification and examples in several languages
- Functional languages: **Haskell** and advanced concepts
- **Stream API** and **lambda-expressions** in Java
- Ownership and Borrowing in **Rust**
- Scripting Languages and **Python**

Programming language pragmatics

- 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*

Inversion of control: unlike in libraries, the overall program's flow of control is not dictated by the caller, but by the framework

Framework Design is a challenging task. It requires mastering of design patterns, OO methods, polymorphism…

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

Main features of RUST

- Introduction to **RUST**
- Avoiding Aliasing + Mutability
- Ownership and Borrowing
- Lifetimes and the Borrow Checker
- Traits, generics, multi-threading
- Unsafe RUST
- Smart Pointers

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**

Design Patterns

Design Patterns in a 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!)

(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:

- 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:

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

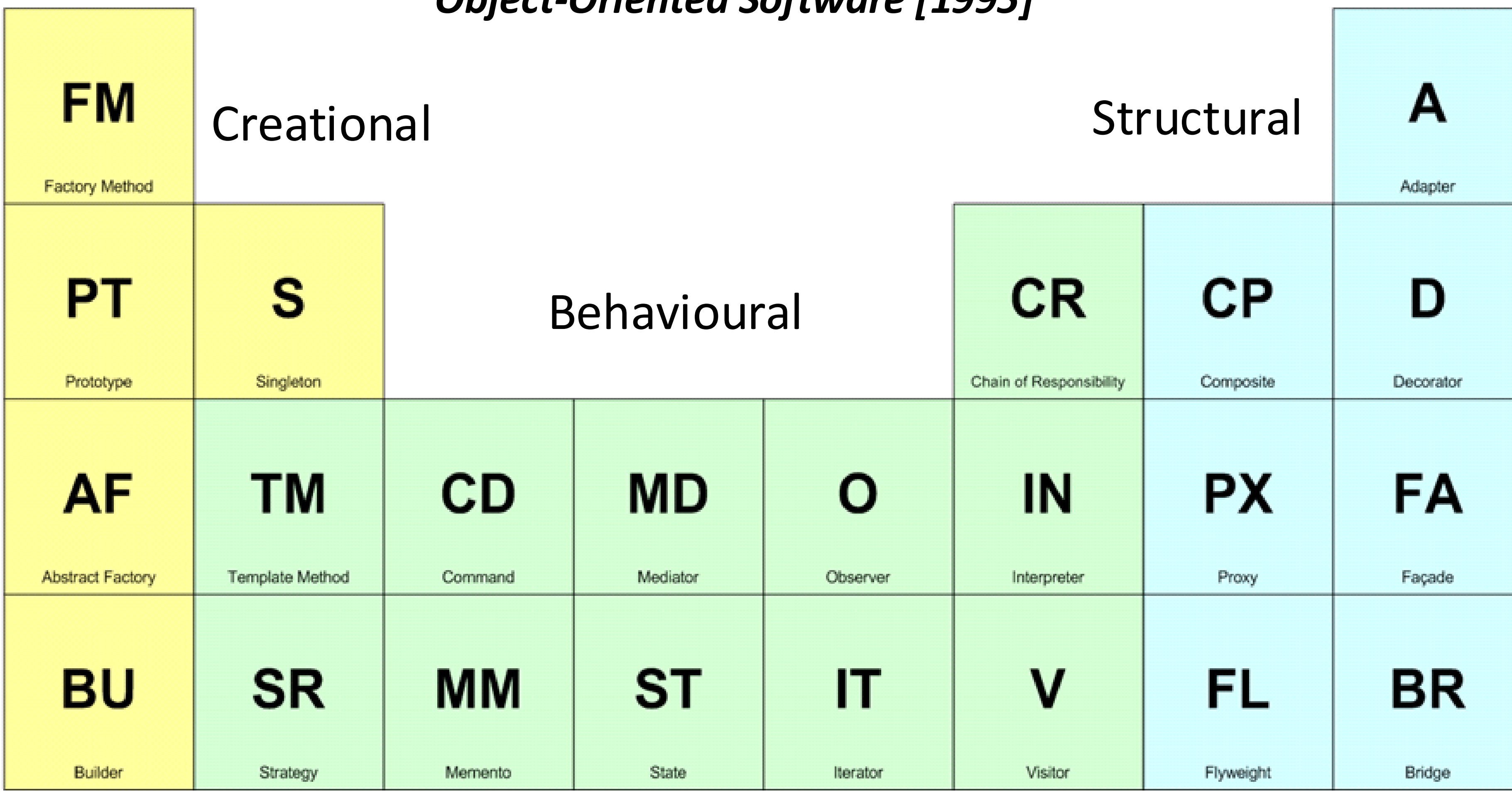
Main components of a Design Pattern

- **Name**: meaningful text that reflects the problem, e.g. ***Bridge***, ***Mediator***, ***Visitor***
- **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!
- **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]



5.5. Pattern: Singleton (Creational)

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.

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

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