Software Design & Code Construction

Software Engineering

Theoretical Concepts

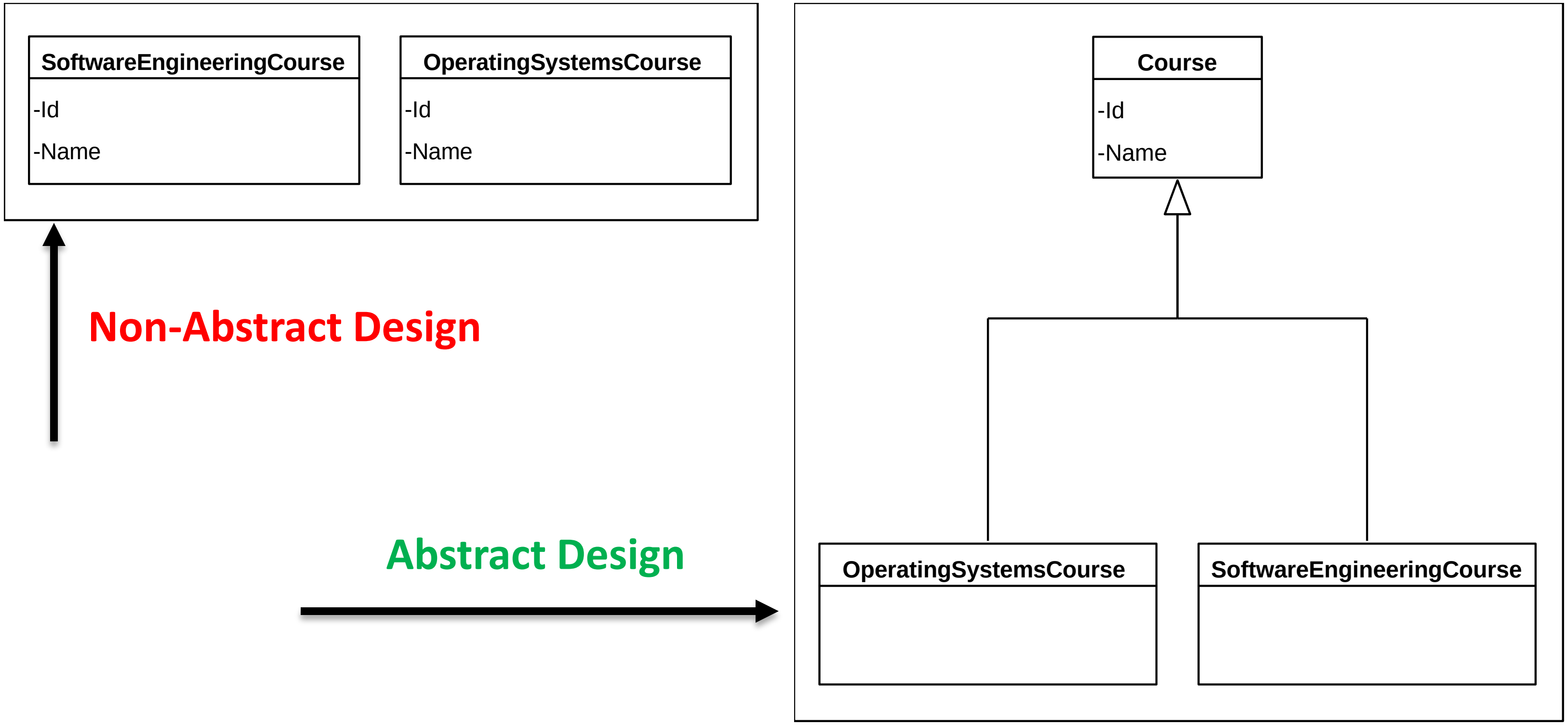
1. Software Design
    1. Definition.
    2. Technical Concepts.
    3. Object Orientation.
    4. Modeling.
2. Code Construction.
    1. Coding Conventions.
        1. Importance.
        2. Naming Conventions.
        3. Best Practices.
    2. Code Appearance and Commenting.
    3. Defensive Programming.

Software Design

- **Software design process** is a sequence of steps that enable the designer to describe all aspects of the software to be built.
- Software design has some **principles** that judge the design from architectural point of view:
  - Design process should consider **alternative approaches**, judging each based on the requirements of the problem.
  - The design should **not reinvent the wheel**.
  - The design should **minimize the intellectual distance** between the software and the problem as it exists in the real world.
  - The design should be **structured** to accommodate change.
  - The design should be assessed for **quality** as it is being created.
  - The design should be reviewed to **minimize** conceptual (**semantic**) **errors**.

Software Design – Concepts

- **Abstraction**: It is the process or result of generalization by reducing the information content of a concept .



SoftwareEngineeringCourse
-Id
-Name

OperatingSystemsCourse
-Id
-Name

Non-Abstract Design

Abstract Design

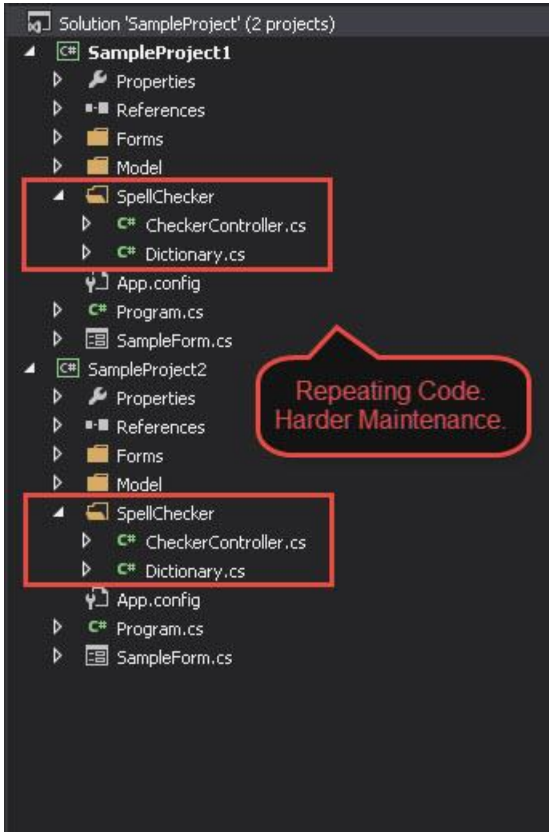
Course
-Id
-Name

OperatingSystemsCourse

SoftwareEngineeringCourse

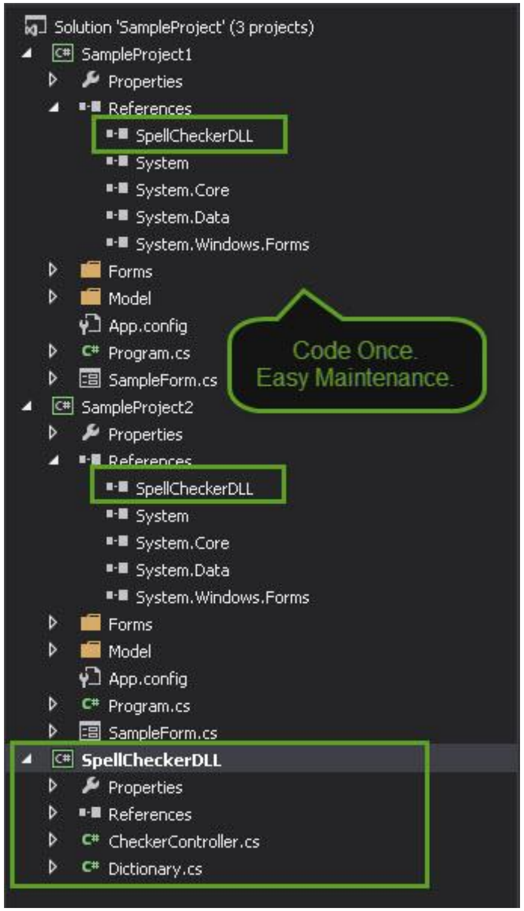
Software Design – Concepts (Cont.)

- **Reusability**: The software is able to add further features and modification with slight or no modification.



```
Solution 'SampleProject' (2 projects)
  SampleProject1
    Properties
    References
    Forms
    Model
    SpellChecker
      CheckerController.cs
      Dictionary.cs
    App.config
    Program.cs
    SampleForm.cs
  SampleProject2
    Properties
    References
    Forms
    Model
    SpellChecker
      CheckerController.cs
      Dictionary.cs
    App.config
    Program.cs
    SampleForm.cs
```

Repeating Code.
Harder Maintenance.



```
Solution 'SampleProject' (3 projects)
  SampleProject1
    Properties
    References
      SpellCheckerDLL
      System
      System.Core
      System.Data
      System.Windows.Forms
    Forms
    Model
    App.config
    Program.cs
    SampleForm.cs
  SampleProject2
    Properties
    References
      SpellCheckerDLL
      System
      System.Core
      System.Data
      System.Windows.Forms
    Forms
    Model
    App.config
    Program.cs
    SampleForm.cs
  SpellCheckerDLL
    Properties
    References
    CheckerController.cs
    Dictionary.cs
```

Code Once.
Easy Maintenance.

Software Design – Concepts (Cont.)

- **Refinement**: It is the process of elaboration.
- **Data Structure**: It is a representation of the logical relationship among individual elements of data.
- **Information Hiding**: Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.
- **Compatibility**: The software is able to operate with other products that are designed for interoperability with another product.
- **Extensibility**: New capabilities can be added to the software without major changes to the underlying architecture.
- **Fault-tolerance**: The software is resistant to and able to recover from component failure.
- **Maintainability**: A measure of how easily bug fixes or functional modifications can be accomplished. High maintainability can be the product of modularity and extensibility.

Software Design – Concepts (Cont.)

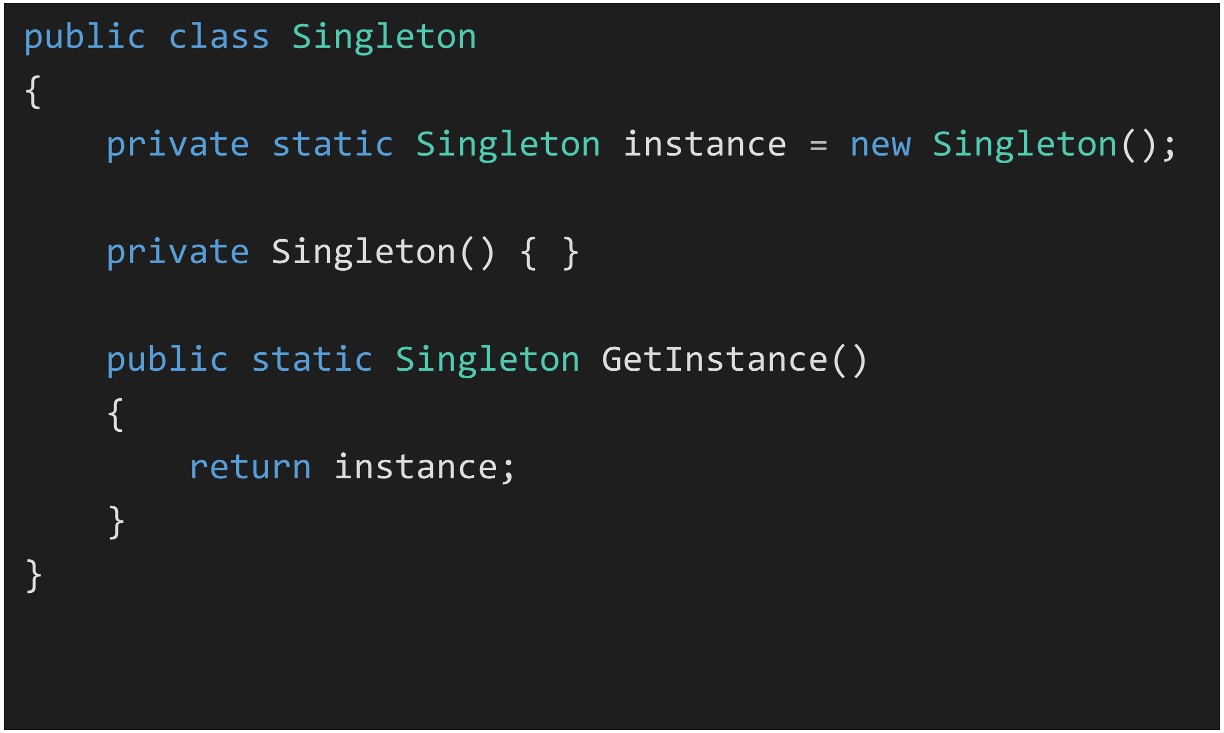
- **Modularity**: The resulting software comprises well defined, independent components. That leads to better maintainability.
- **Reliability**: The software is able to perform a required function under stated conditions for a specified period of time.
- **Robustness**: The software is able to operate under stress or tolerate unpredictable or invalid input.
- **Security**: The software is able to withstand hostile acts and influences.
- **Usability**: The software user interface must be usable for its target user/audience.
- **Performance**: The software performs its tasks within a user-acceptable time. The software does not consume too much memory.
- **Portability**: The usability of the same software in different environments.
- **Scalability**: The software adapts well to increasing data or number of users.

Software Design – Modeling

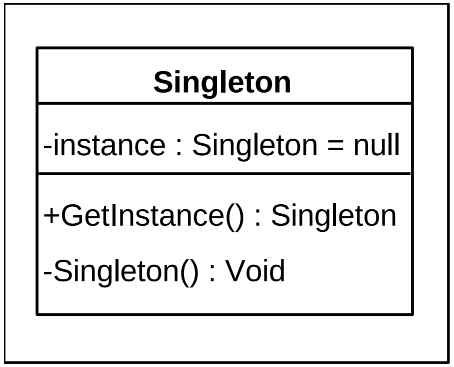
- **Software modeling** is the process of creating models that describe the software in terms of **structure** and **procedures**.
- Unified Modeling Language (UML)
  - **Definition**: It is a modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system.
  - It uses **standard** visual **artifacts** to describe these **diagrams**:
    - **Use case** diagram.
    - **Component** diagram.
    - **Class** diagram.
    - **Activity** diagram.
    - **Sequence** diagram.
    - And others...
  - Using UML standard simplifies:
    - The process of **understanding** the software even for **non technical** users.
    - **Maintenance** of the design as it uses a standard techniques known by most of software engineers.
  - **CASE tools** are the set of tools and methods to a software system with the desired end result of high-quality, defect-free, and maintainable software products.
  - Sample of software design CASE tools are UML studios (e.g. **Visual Paradigm**).

Design Patterns

- **Singleton**: It is a design pattern that restricts the instantiation of a class to one object.



```
public class Singleton
{
    private static Singleton instance = new Singleton();

    private Singleton() { }

    public static Singleton GetInstance()
    {
        return instance;
    }
}
```



Singleton

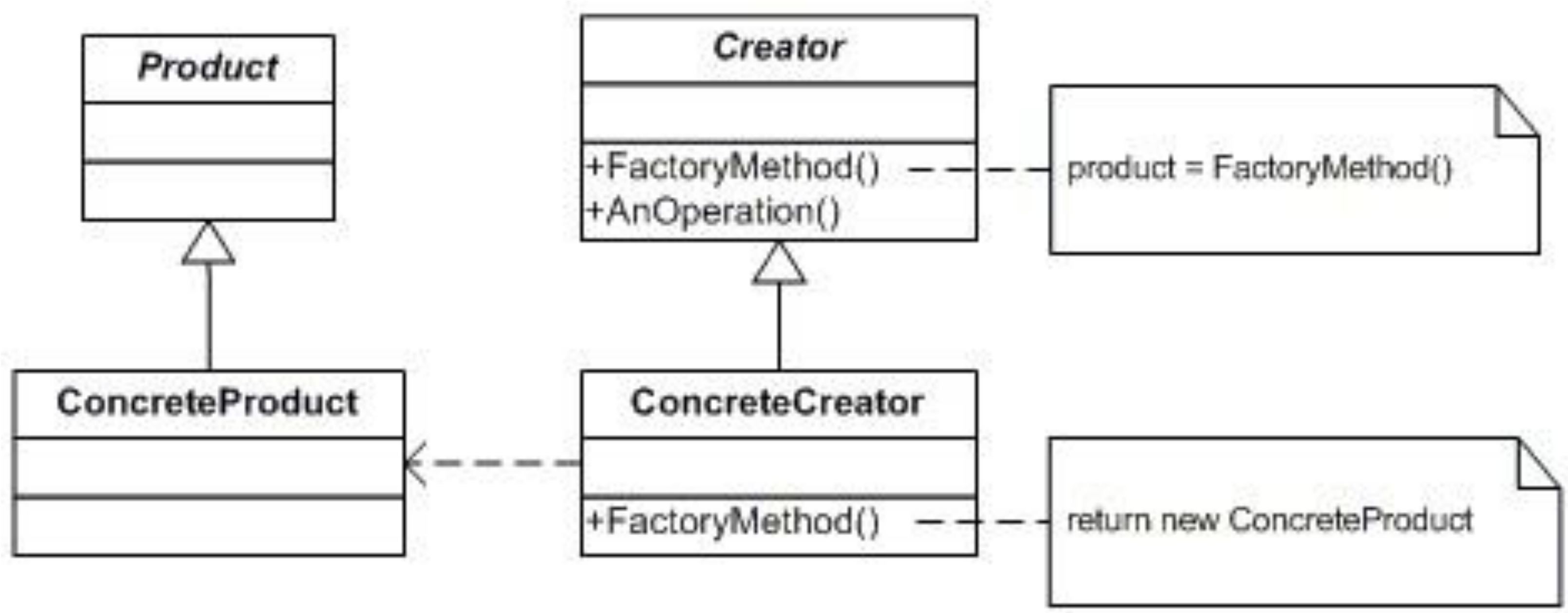
-instance : Singleton = null

+GetInstance() : Singleton

-Singleton() : Void

Design Patterns (Cont.)

- **Factory Design Pattern**: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



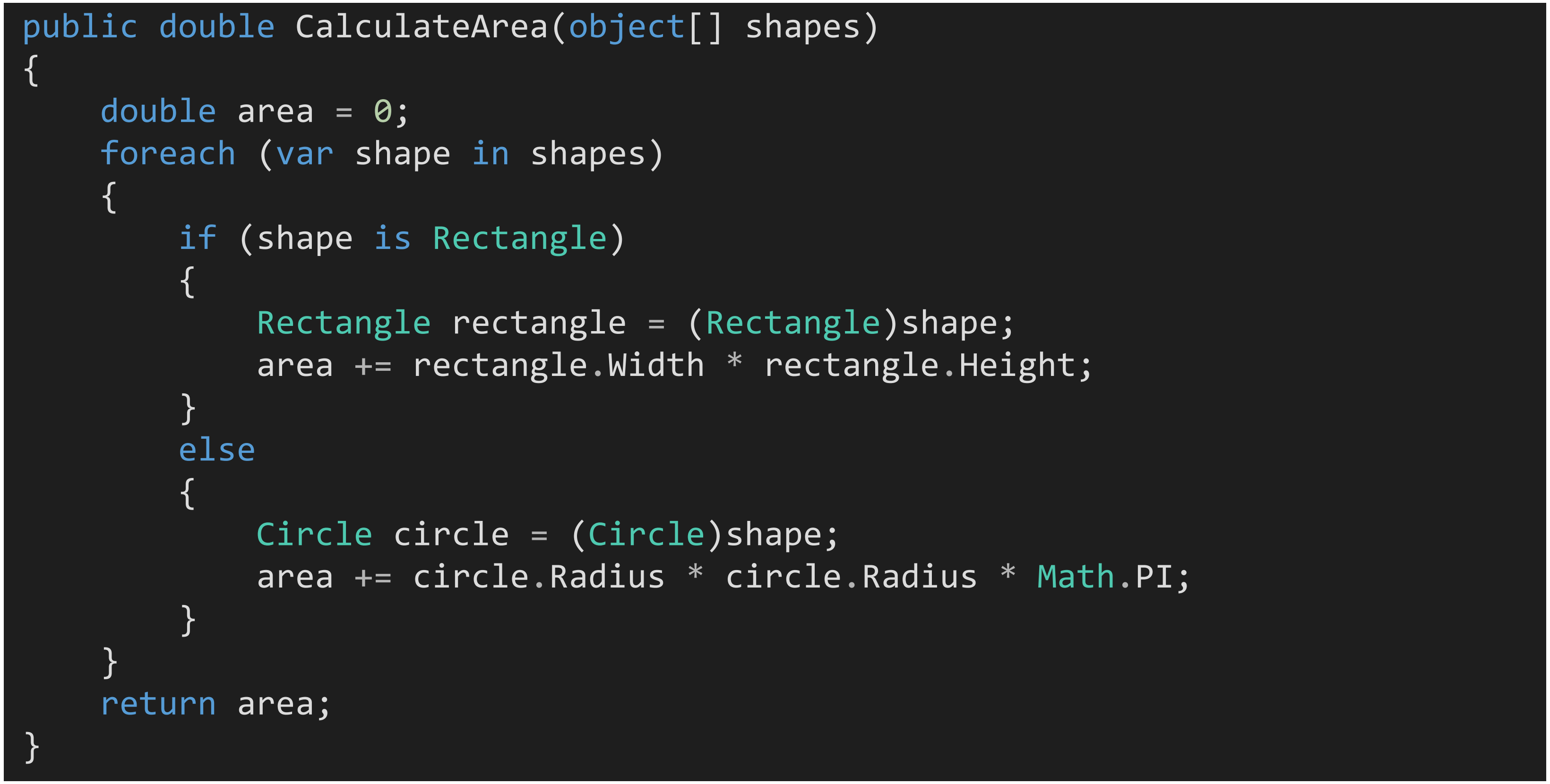
Code Construction – Definition

- **Code construction** is the principles and disciplines used to write a **well-structured maintainable code**.
- It concentrates on **coding styles** and **conventions** that are used to deliver a beautiful program code.
- It applies on any software coding process whatever the programming language is.

Code Construction – Principles

1. **Closed for modifications and open for extensions**: the code entities (classes, functions and modules) should allow its behavior to be extended without modifying its source code.

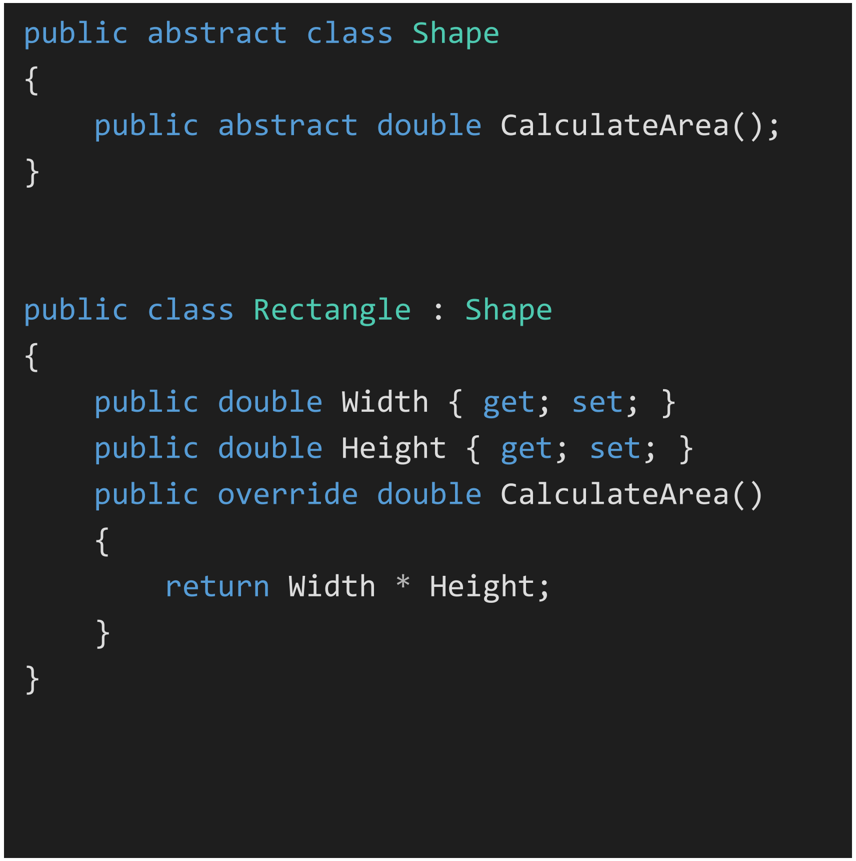
Code Opened for modifications



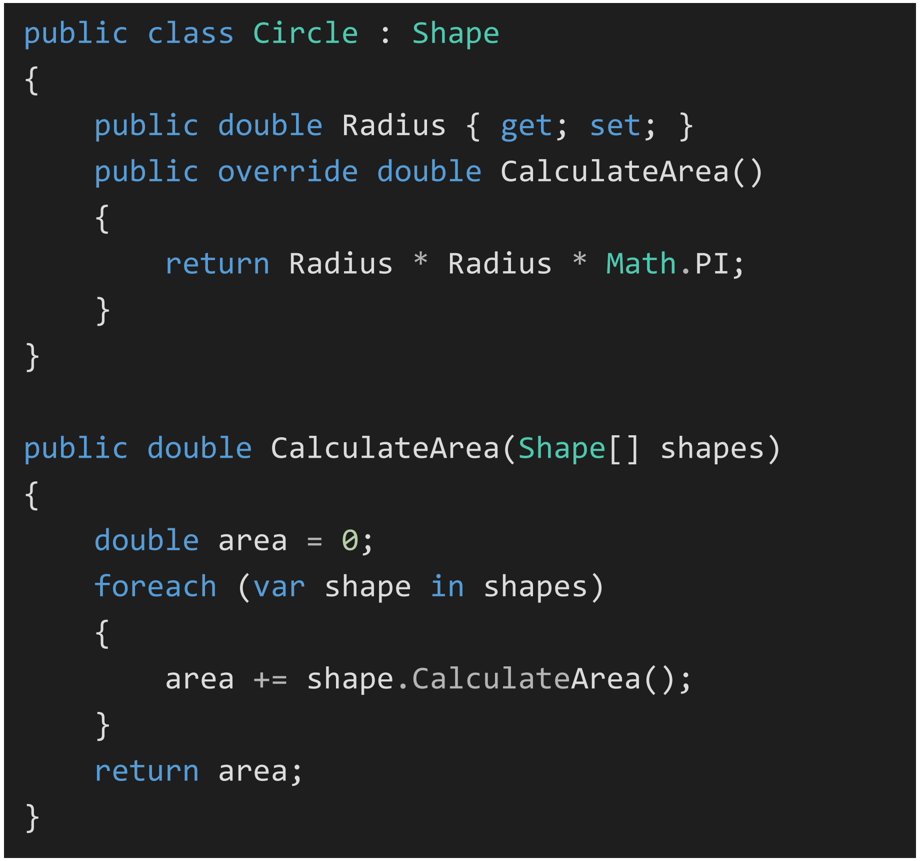
```
public double CalculateArea(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle)shape;
            area += rectangle.Width * rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }
    return area;
}
```

Code Construction – Principles

Closed for modifications and open for extensions



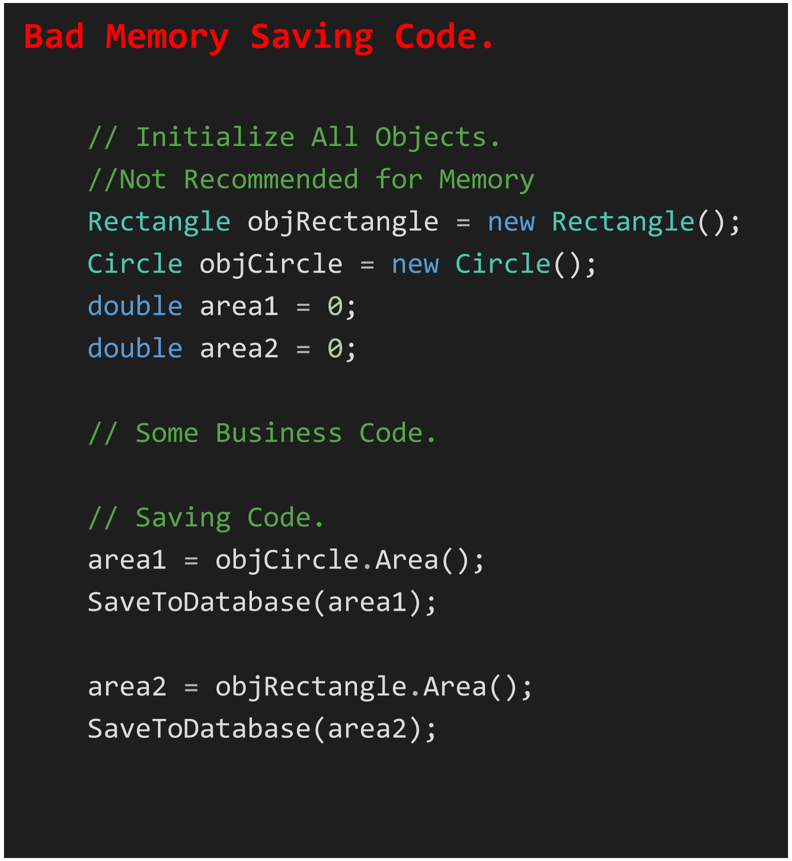
```
public abstract class Shape
{
    public abstract double CalculateArea();
}


public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double CalculateArea()
    {
        return Width * Height;
    }
}
```



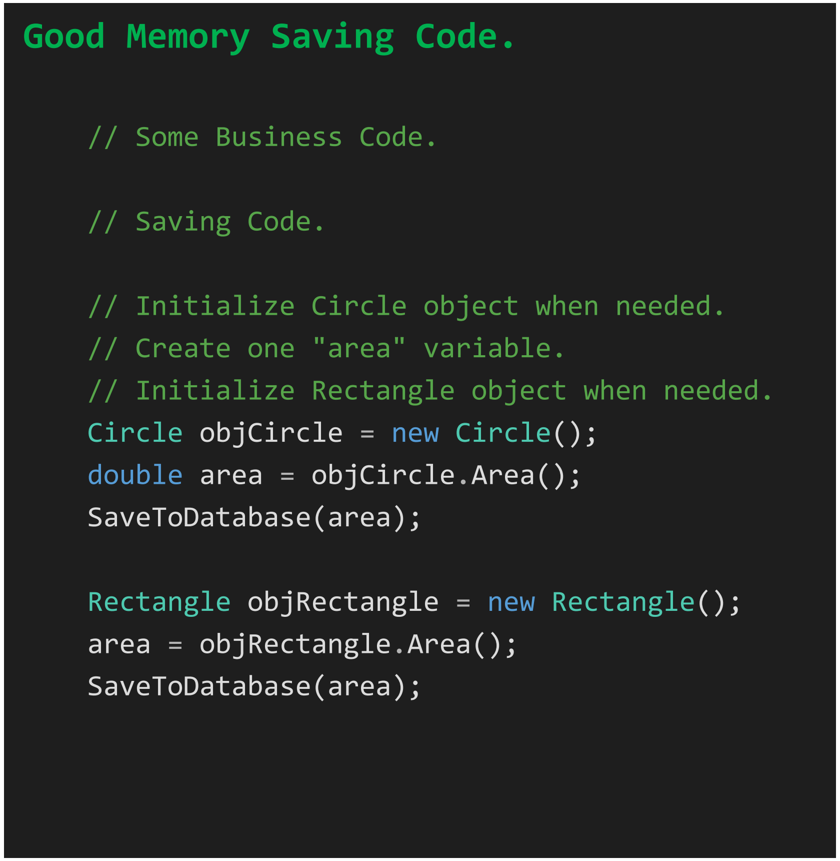
```
public class Circle : Shape
{
    public double Radius { get; set; }
    public override double CalculateArea()
    {
        return Radius * Radius * Math.PI;
    }
}

public double CalculateArea(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.CalculateArea();
    }
    return area;
}
```

Code Construction – Principles (Cont.)

1. **Saving memory**: by handling objects allocation and de-allocation especially on low memory devices (e.g. mobile phones and embedded systems).



```
Bad Memory Saving Code.

    // Initialize All Objects.
    //Not Recommended for Memory
    Rectangle objRectangle = new Rectangle();
    Circle objCircle = new Circle();
    double area1 = 0;
    double area2 = 0;

    // Some Business Code.

    // Saving Code.
    area1 = objCircle.Area();
    SaveToDatabase(area1);

    area2 = objRectangle.Area();
    SaveToDatabase(area2);
```



```
Good Memory Saving Code.

    // Some Business Code.

    // Saving Code.

    // Initialize Circle object when needed.
    // Create one "area" variable.
    // Initialize Rectangle object when needed.
    Circle objCircle = new Circle();
    double area = objCircle.Area();
    SaveToDatabase(area);

    Rectangle objRectangle = new Rectangle();
    area = objRectangle.Area();
    SaveToDatabase(area);
```

Code Construction – Principles (Cont.)

1. **Saving space**: the code should take care of the data size and keep it optimized as possible.
2. **Saving bandwidth**: the code should take care of bandwidth usage and use it carefully.
3. **Performance**: the most important factor in determining code quality is the performance and how fast it achieves the targeted tasks without hanging of taking too much time.
4. **Avoid code duplication**: code duplication is a sequence of source code that occurs more than once, either within a program or across different programs owned or maintained by the same entity. Inappropriate code duplication increases maintenance costs both in time and money.

Coding Conventions – Importance

- **Coding conventions** are a set of **guidelines** for a specific programming language that recommend programming style.
- Importance:
  - **40%–80%** of the lifetime **cost** of a piece of software goes to **maintenance**.
  - Hardly any software is maintained for its whole life by the original author.
  - Code conventions improve the **readability** of the software, allowing engineers to **understand new code** more **quickly** and thoroughly.
  - If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Coding Conventions – Naming Conventions

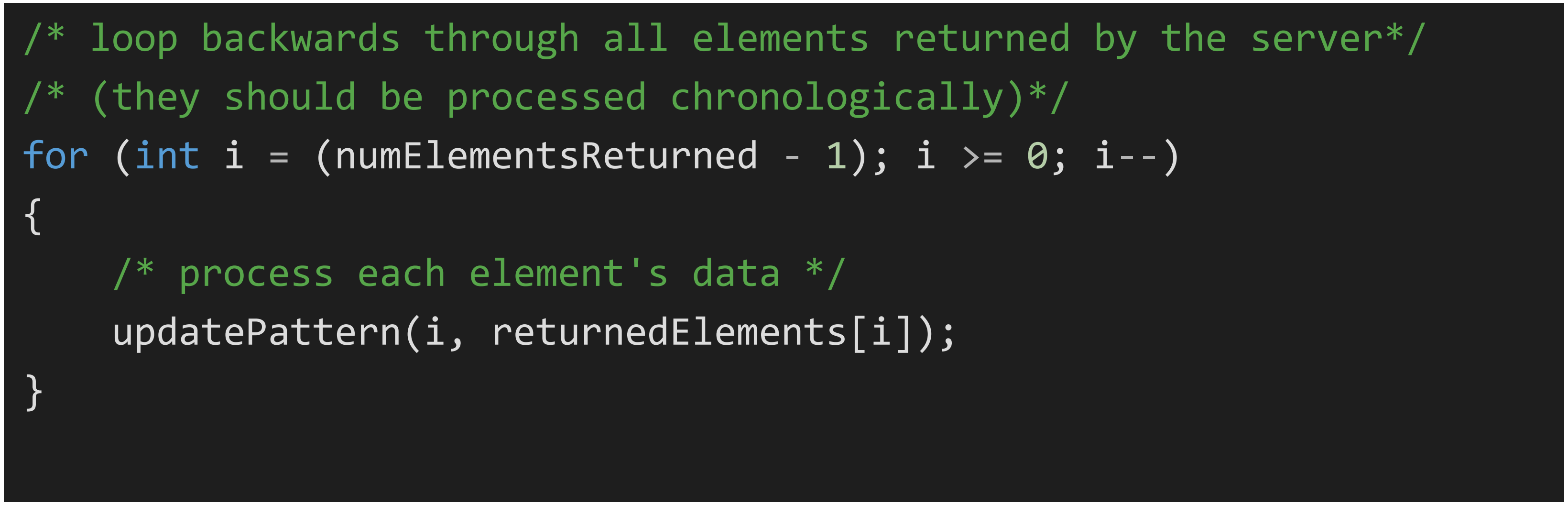
- **Naming conventions** are the set of rules for choosing the character sequence to be used for identifiers which denote variables, types, functions, and other entities in source code and documentation.
- Best practices:
  - Using **verbs** for **functions**.
  - Using **nouns** for **classes**.
  - Using **Pascal cased naming** for **class names**, **functions names**, **properties** (e.g. CarManager).
  - Using **camel cased** naming for **local variables** names (e.g. carCounter).
  - Names choice guidelines:
    - **Descriptive** names for what the function is doing.
  - Sample code with naming conventions (next slide).

Coding Conventions – Naming Conventions (Cont.)

```
// "CalculateArea" Function Name: Pascal Cased.
// "CalculateArea" Function Name: Descriptive Verb.
public double CalculateArea(object[] shapes)
{
    double calculatedArea = 0; // "calculatedArea" Local Variable: Camel Cased.
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            // "Rectangle" and "Circle" Classes' Names: Nouns.
            // "Rectangle" and "Circle" Classes' Names: Pascal Cased.
            Rectangle rectangle = (Rectangle)shape;
            // "Height" and "Width" Properties of "rectangle" object: Camel Cased.
            calculatedArea += rectangle.Width * rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            // "Radius" Property of "circle" object: Camel Cased.
            calculatedArea += circle.Radius * circle.Radius * Math.PI;
        }
    }
    return calculatedArea;
}
```

Code Commenting

- **Comments Types**
  - **Planning and reviewing**: used to outline intention prior to writing the actual code. In this case, it should explain the logic behind the code rather than the code itself.



```
/* loop backwards through all elements returned by the server*/
/* (they should be processed chronologically)*/
for (int i = (numElementsReturned - 1); i >= 0; i--)
{
    /* process each element's data */
    updatePattern(i, returnedElements[i]);
}
```

Code Commenting (Cont.)

- **Code description**: used to summarize code or to explain the programmer's intent.
  - Good comments do not repeat the code or explain it. They clarify its intent.
  - Do not document bad code – rewrite it.

```
// Reload local settings again because of server errors produced when reuse form data.
// No documentation available on server behavior issue, so just coding around it.
settings = server.Load("local settings");
```

- **Algorithmic description**: used to explain new or difficult algorithms used to solve specific problems.

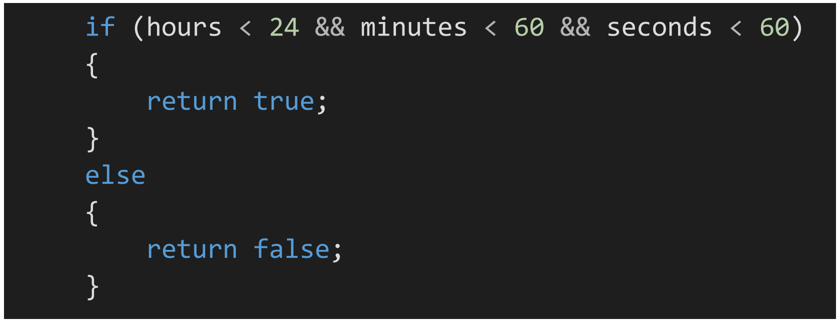
```
List<string> list = new List<string>() { "b", "b", "c", "d", "a"};
// Need a stable sort. Besides, the performance really does not matter.
DoInsertionSort(list);
```

- **Debugging**: used to comment out a code snippet that will not be executed in the final program.

```
if (options.equals("e"))
    optionEnabled = true;
/*
if (options.equals("d"))
    optionDebug = true;
 */
if (options.equals("v"))
    optionVerbose = true;
```

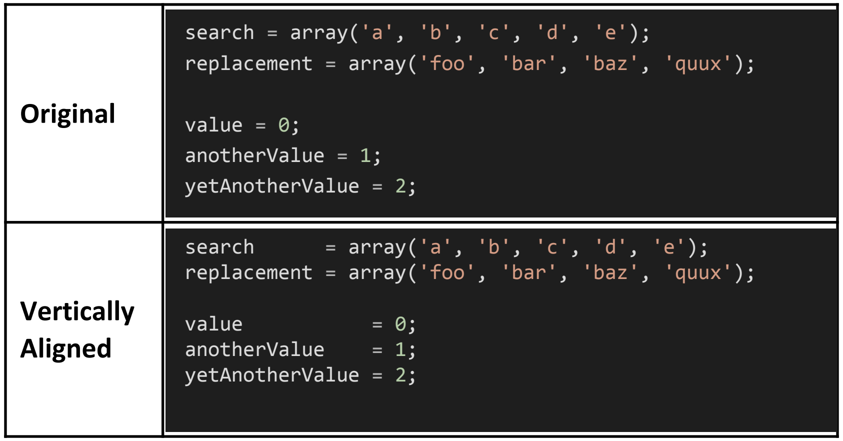
Code Appearance

- Code appearance guidelines
  - Indentation alignment.



```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
    return false;
}
```

  - Vertical Alignment.



Original

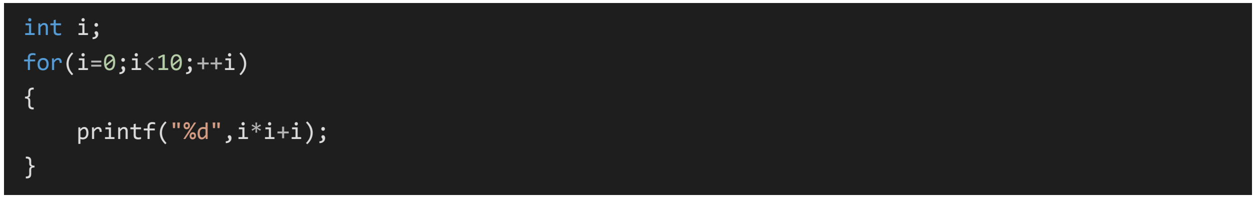
```
search = array('a', 'b', 'c', 'd', 'e');
replacement = array('foo', 'bar', 'baz', 'quux');

value = 0;
anotherValue = 1;
yetAnotherValue = 2;
```

Vertically Aligned

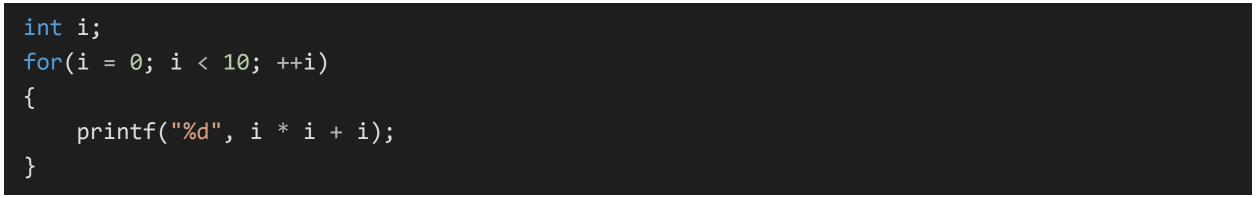
```
search      = array('a', 'b', 'c', 'd', 'e');
replacement = array('foo', 'bar', 'baz', 'quux');

value           = 0;
anotherValue    = 1;
yetAnotherValue = 2;
```

Code Appearance (Cont.)

- Code appearance guidelines
  - Spaces and tabs standards.

<table>
<tr>
<td><b>Original</b></td>
<td>


```c
int i;
for(i=0;i<10;++i)
{
    printf("%d",i*i+i);
}
```

</td>
</tr>
<tr>
<td><b>With Spaces</b></td>
<td>


```c
int i;
for(i = 0; i < 10; ++i)
{
    printf("%d", i * i + i);
}
```

</td>
</tr>
</table>

  - Functions length best practice. (e.g. With maximum of 200 lines)
  - Line length best practice. (e.g. 80 character on average)

Defensive Programming

- It is a form of **defensive** design intended to ensure the continuing function of a piece of software under **unforeseen circumstances**.
- **Input validation**:
  - Check the values of all data from external sources.
  - Check the values of all routine input parameters.
  - Design how to handle bad inputs.
- **Exceptions handling**:
  - Use exceptions to notify other parts of the program about errors that should not be ignored.
  - Throw an exception only for conditions that are truly exceptional.

Beautiful Coding Aspects

- **Maintainability**: the code should be easy to change and maintain. (e.g. standard function length should be followed to make it more understandable and hence more maintainable).
- **Extensibility**: the code should take into consideration the future growth. (e.g. function's parameters should be well encapsulated to avoid function interface change when it is needed to be extended).
- **Portability**: the code should run on different platforms and localizations. (e.g. no hard codded values in the code, use resources localized values, and configurations items instead).
- **Reusability**: the implementation should be generic to enable reusing it in different modules.
- **Loose coupling**: no code component (e.g. class, function) should have strong coupling with another component, where a function calling another function should not be affected when some local variable in the other function has changed.

Questions

1. What is the software design?
2. List the software design aspects.
3. What is information hiding? And how will it affect the quality of the software?
4. What is the difference between fault-tolerance and maintainability?
5. What is code construction?
6. List code construction principles.
7. Why code review is important in the coding cycle?
8. What are the best practices of naming conventions?
9. What is the defensive programming?
10. List two techniques of defensive programming?
11. What are the comments types?
12. List beautiful code aspects.
13. List three items affecting the code appearance.

Thanks!