

Learning Objectives

1. Define object-oriented programming
2. Position object-oriented programming within Software Development Lifecycle (SDLC)
3. Review object-oriented languages and tools
4. Demonstrate object-oriented programming concepts with example
5. Distinguish between a class and an object
6. Identify and define “six” object-oriented concepts
7. Identify the superclass and the subclass in an inheritance relationship
8. Demonstrate inheritance, ownership, and abstraction in snippets of Java code
9. Distinguish between aggregation and composition
10. Depict classes and their relationships using UML class diagrams
11. Explain common design patterns
12. Define and demonstrate the common software patterns
13. Identify and describe characteristics of bad software: rigidity, immobility, fragility
14. Describe how object-oriented programming is fundamentally different
15. Justify the choice to use an object-oriented approach in developing software

You will notice very quickly that I prefer examples and actual source code over more philosophical discussion. Please feel free to ask for more development philosophy and background if you desire more.

Define Object-Oriented Programming [\[Link\]](#)

Object-oriented programming (OOP) is a programming model based on the concept of "objects", which contain both Attributes (data) and Methods (procedures) that operate on those attributes.

Most popular OOP languages are class-based, meaning that objects are instances of classes.

It includes concepts, patterns, and principles for designing and implementing modern software products.

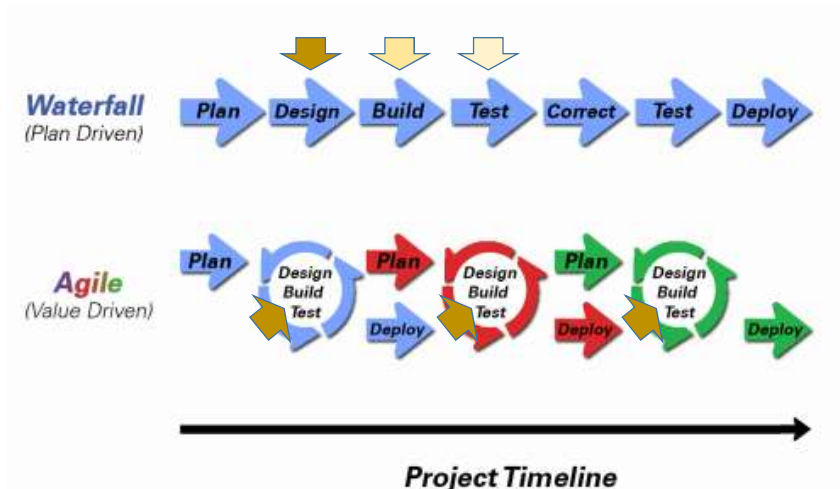
I often start with definitions from Wikipedia and other sources.

Concepts – powerful features that prove indispensable to modern software development, brought to us automatically by object-oriented programming.

Patterns – tried-and-true templates for forging relationships between classes

Principles – guidelines that help you determine what classes are needed and how they should divide up the work

Position Object-Oriented Programming within Various Development Methodologies



Development Methodology and Software Development Lifecycle (SDLC) are often used interchangeably.

The Iterative development methodology is not depicted here as even the mainstays and inventors of the Iterative development methodology seem to be moving toward agile. Plus as Waterfall “holdouts” move, they seem to be moving directly toward Agile.

Development Methodologies (SDLCs) are a future Bonus Topic. There is a optional slide and notes at the end of this deck.

Object oriented programming practices evolve and reprioritize depending on the development

In Waterfall (as well as in Iterative) object-oriented design often play a critical role in the (big upfront) design activities. UML diagrams and project artifacts are often important to the overall project success.

Practical reality has been that these design artifacts often do not reflect the actual implementation and are rarely maintained or updated.








The Agile practitioners do not reject these design artifacts. However, the focus on shorter time horizons, evolving architecture, and working code changes the value proposition for object-oriented practices to more focus on the build, test, enhance activities.

More later on Development Methodologies

Review object-oriented languages and tools

The TIOBE index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The full TIOBE is available online [\[Link\]](#).

TIOBE Index for March 2017:

Mar 2017	Mar 2016	Change	Programming Language	Ratings	Change
1	1		Java	16.384%	-4.14%
2	2		C	7.742%	-6.86%
3	3		C++	5.184%	-1.54%
4	4		C#	4.409%	+0.14%
5	5		Python	3.919%	-0.34%
6	7		Visual Basic .NET	3.174%	+0.61%
7	6		PHP	3.009%	+0.24%
8	8		JavaScript	2.667%	+0.33%
9	11		Delphi/Object Pascal	2.544%	+0.54%
10	14		Swift	2.268%	+0.68%

We will utilize mostly Java and C# for our object-oriented programming examples. We may or may not do any Python work. Since it is often 'unnatural' to show procedural programming examples in Java, C#, or Python, we will implement programs in C to demonstrate procedure programming examples.

Note that our reluctance to utilize C++ as a OOP learning tool is does not diminish the value of the C++ toolset. However, C++ is generally considered a very powerful set of tools with a steep learning curve. It's a sharp knife... use it carefully.

Demonstrate object-oriented programming concepts with example

The body mass index (BMI) is a statistic developed by Adolphe Quetelet in the 1900's for evaluating body mass. It is not related to gender and age. It uses the same formula for men as for women and children.

The body mass index is calculated based on the following formula:

$$\text{BMI} = \text{weight [kg]} / (\text{height [m]} * \text{height [m]})$$

Procedural BMI (body mass index):

Data:

- Height
- Weight

Procedures (or functions):

- CalcBMI

Object-Oriented BMI:

Class BMI

- Attributes
 - Height
 - Weight
- Methods
 - CalcBMI

I often start with definitions from Wikipedia and other sources.

Example: Procedural vs. Object Oriented Programming

Procedural BMI (C):

```
// BMI Calculator (Procedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; //6'1"
    weight = 190.0; // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

Object Oriented BMI (Java):

```
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

Do you see the logic issue? Hang onto that for a moment.



Distinguish between a class and an object

Procedural BMI:

```
// BMI Calculator (Procedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; //6'1"
    weight = 190.0;             // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

Object Oriented BMI:

```
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

Do you see the logic issue? Hang onto that for a moment.



Distinguish between a class and an object

Procedural BMI:

```
// BMI Calculator (Procedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; //6'1"
    weight = 190.0;             // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

Object Oriented BMI:

```
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

Do you see the logic issue? Hang onto that for a moment.

Week 1 Outcomes

1. Define object-oriented programming
2. Distinguish between a class and an object
3. Identify and define “six” object-oriented concepts
4. Identify the superclass and the subclass in an inheritance relationship
5. Demonstrate inheritance, ownership, and abstraction in snippets of Java code
6. Distinguish between aggregation and composition
7. Describe how object-oriented programming is fundamentally different
8. Justify the choice to use an object-oriented approach in developing software
9. Define and demonstrate the common software patterns
10. Depict classes and their relationships using UML class diagrams
11. Explain common design patterns
12. Identify and describe characteristics of bad software: rigidity, immobility, fragility

The Problem? *BMI formula assumes M (height) and KG (weight)*

Procedural BMI:

```
// BMI Calculator (Procedural C)
// BMI = weight over height squared

float height = 0;
float weight = 0;

float CalcBMI(void) {
    return weight / (height * height);
}

int main() {
    height = (6.0 * 12.0) + 1.0; //6'1"
    weight = 190.0;              // 190 lbs
    float BMI = CalcBMI();

    printf("BMI: %f\n", BMI);
    return 0;
}
```

Object Oriented BMI:

```
// BMI Calculator (OOP Java)
// BMI = weight over height squared

class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

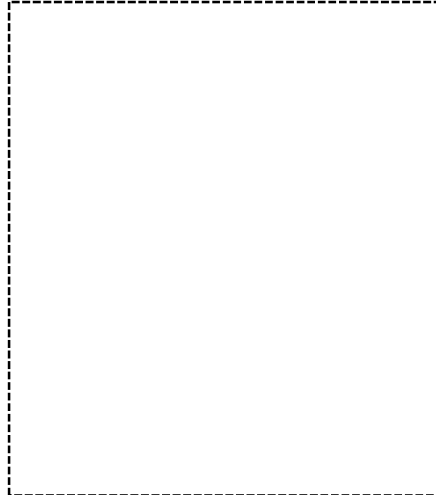
Do you see the logic issue? Hang onto that for a moment.

Revising BMI Implementations

Procedural BMI (revised):

```
*****  
// BMI Calculator (Procedural C)  
// BMI = weight over height squared  
  
float heightinches = 0;  
float weightinlbs = 0;  
  
float heightinm = 0;  
float weightinkg = 0;  
  
float CalcBMIMetric(void) {  
    return weightinkg / (heightinm * heightinm);  
}  
  
float CalcBMIEnglish(void) {  
    heightinm = heightinches * 0.025;  
    weightinkg = weightinlbs * 0.45;  
    return CalcBMIMetric();  
}  
  
int main() {  
    heightinches = (6.0 * 12.0) + 1; // 6'1"  
    weightinlbs = 190.0;           // 190 lbs  
  
    float BMI = CalcBMIEnglish();  
    printf("BMI: %f\n", BMI);  
    return 0;  
}
```

Object Oriented BMI (revised):



Now consider what it would be like to reuse this C procedural code. Important note: Object Oriented program is NOT necessarily optimized for writing new code. It IS optimized for reusing, supporting, testing, and enhancing code! How would this change if instead of ~20 lines of code, we have a thousand lines of code... or 10,000... or 1,000,000 lines? Software development and testing complexity grows exponentially as the size of the code grows.

Consider: : rigidity, immobility, fragility within the ongoing sdlc (software/systems development lifecycle)

Waterfall, Iterative, and Agile

Identify and define “six” (3plus) object-oriented concepts

Object-oriented concepts:

1. **Encapsulation...** and Information Hiding
2. **Inheritance...** and Abstraction
3. **Polymorphism**

Plus... Composition & Aggregation

Encapsulation

Object Oriented BMI:

```
/**
 * BMI Calculator (OOP Java)
 * BMI = weight over height squared
 */
class BMI {
    public float height = 0;
    public float weight = 0;

    public float CalcBMI() {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        myBMI.height = (float)((6.0 * 12.0) + 1.0); //6'1"
        myBMI.weight = (float)190.0; //190 lbs
        float BMIresult = myBMI.CalcBMI();

        System.out.println(BMIresult);
    }
}
```

Encapsulated Object Oriented BMI:

```
/**
 * BMI Calculator (OOP Java)
 * BMI = weight over height squared
 */
class BMI {
    public float CalcBMI(float height, float weight) {
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMI();
        float BMIresult = myBMI.CalcBMI(
            (float)((6.0 * 12.0) + 1.0) /*height*/,
            (float)190.0 /*weight*/);

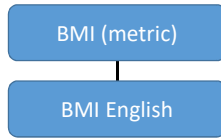
        System.out.println(BMIresult);
    }
}
```

Consider: How would you add protective code around setting height to 0 in the first option? ... in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines of code.

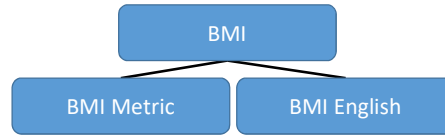
Encapsulation is a feature of nearly all modern development languages... not just object-oriented languages.

Inheritance Options for Implement English BMI Units

Option #1:

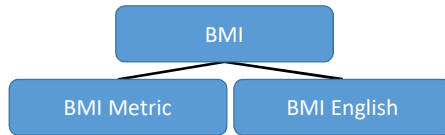


Option #2:



Remember: Current implementation does not work with English units.

Inheritance to implement English units... And Abstraction



```
/*=====
// BMI Calculator (OOP Java)
// BMI = weight over height squared

abstract class BMI {
    abstract public float CalcBMI(float height, float weight);
}

class BMIMetric extends BMI {
    public float CalcBMI(float height, float weight) {
        return weight / (height * height);
    }
}

class BMIEnglish extends BMI {
    public float CalcBMI(float height, float weight) {
        // Convert to meters and kg.
        height = height * (float)0.025;
        weight = weight * (float)0.45;
        return weight / (height * height);
    }
}

class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMIEnglish();
        float BMIresult = myBMI.CalcBMI(
            (float)((6.0 * 12.0) + 1.0) /*height*/,
            (float)190.0 /*weight*/);

        System.out.println(BMIresult);
    }
}
```

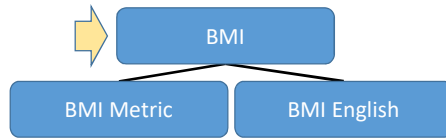
Consider: How would you add protective code around setting height to 0 in the first option? ... in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines off code.

Encapsulation is a feature of nearly all modern development languages... not just object-oriented languages.

Week 1 Outcomes

1. Define object-oriented programming
2. Distinguish between a class and an object
3. Identify and define “six” object-oriented concepts
4. Identify the superclass and the subclass in an inheritance relationship
5. Demonstrate inheritance, ownership, and abstraction in snippets of Java code
6. Distinguish between aggregation and composition
7. Describe how object-oriented programming is fundamentally different
8. Justify the choice to use an object-oriented approach in developing software
9. Define and demonstrate the common software patterns
10. Depict classes and their relationships using UML class diagrams
11. Explain common design patterns
12. Identify and describe characteristics of bad software: rigidity, immobility, fragility

Identify the superclass and the subclass in an inheritance relationship



➡ Superclass

```
/*=====
// BMI Calculator (OOP Java)
// BMI = weight over height squared

abstract class BMI {
    abstract public float CalcBMI(float height, float weight);
}

class BMIMetric extends BMI {
    public float CalcBMI(float height, float weight) {
        return weight / (height * height);
    }
}

class BMIEnglish extends BMI {
    public float CalcBMI(float height, float weight) {
        // Convert to meters and kg.
        height = height * (float)0.025;
        weight = weight * (float)0.45;
        return weight / (height * height);
    }
}

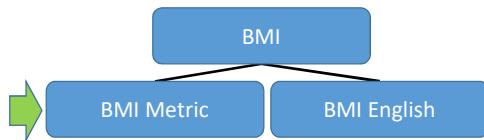
class CalcBMI {
    public static void main(String[] args) {
        BMI myBMI = new BMIEnglish();
        float BMIresult = myBMI.CalcBMI(
            (float)((6.0 * 12.0) + 1.0) /*height*/,
            (float)190.0 /*weight*/);

        System.out.println(BMIresult);
    }
}
```

Consider: How would you add protective code around setting height to 0 in the first option? ... in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines off code.

Encapsulation is a feature of nearly all modern development languages... not just object-oriented languages.

Identify the superclass and the subclass in an inheritance relationship



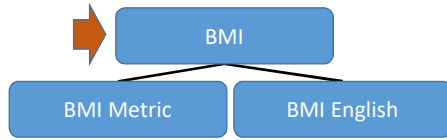
➡ Subclass

```
/* *****  
// BMI Calculator (OOP Java)  
// BMI = weight over height squared  
***** */  
  
abstract class BMI {  
    abstract public float CalcBMI(float height, float weight);  
}  
  
class BMIMetric extends BMI {  
    public float CalcBMI(float height, float weight) {  
        return weight / (height * height);  
    }  
}  
  
class BMIEnglish extends BMI {  
    public float CalcBMI(float height, float weight) {  
        // Convert to meters and kg.  
        height = height * (float)0.025;  
        weight = weight * (float)0.45;  
        return weight / (height * height);  
    }  
}  
  
class CalcBMI {  
    public static void main(String[] args) {  
        BMI myBMI = new BMIEnglish();  
        float BMIresult = myBMI.CalcBMI(  
            (float)((6.0 * 12.0) + 1.0) /*height*/,  
            (float)190.0 /*weight*/);  
  
        System.out.println(BMIresult);  
    }  
}
```

Consider: How would you add protective code around setting height to 0 in the first option? ... in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines off code.

Encapsulation is a feature of nearly all modern development languages... not just object-oriented languages.

Identify the superclass and the subclass in an inheritance relationship



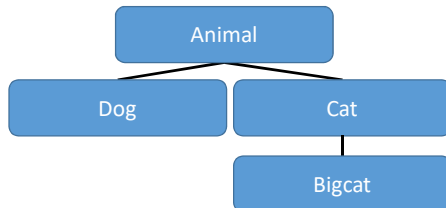
➡ Abstraction

```
/* *****  
// BMI Calculator (OOP Java)  
// BMI = weight over height squared  
***** */  
  
abstract class BMI {  
    abstract public float CalcBMI(float height, float weight);  
}  
  
class BMIMetric extends BMI {  
    public float CalcBMI(float height, float weight) {  
        return weight / (height * height);  
    }  
}  
  
class BMIEnglish extends BMI {  
    public float CalcBMI(float height, float weight) {  
        // Convert to meters and kg.  
        height = height * (float)0.025;  
        weight = weight * (float)0.45;  
        return weight / (height * height);  
    }  
}  
  
class CalcBMI {  
    public static void main(String[] args) {  
        BMI myBMI = new BMIEnglish();  
        float BMIresult = myBMI.CalcBMI(  
            (float)((6.0 * 12.0) + 1.0) /*height*/,  
            (float)190.0 /*weight*/);  
  
        System.out.println(BMIresult);  
    }  
}
```

Consider: How would you add protective code around setting height to 0 in the first option? ... in the second encapsulated example? Once again consider reuse, testing, and additional modification in thousands of lines off code.

Encapsulation is a feature of nearly all modern development languages... not just object-oriented languages.

Polymorphism



⇒ Polymorphic

```
import java.util.Random;

abstract class Animal {
    abstract public void PrintYourAngrySound() {
        Grrrr... Hisssss!
    }
}

class Dog extends Animal {
    public void PrintYourAngrySound() {
        System.out.println("Bark!");
    }
}

class Cat extends Animal {
    public void PrintYourAngrySound() {
        System.out.println("Hisssss!");
    }
}

class Bigcat extends Cat {
    public void PrintYourAngrySound() {
        System.out.println("Grrrr... Hisssss!");
    }
}

class PolyAnimalSounds {
    public static void main(String[] args) {
        Random rand = new Random();

        Animal someAnimal = new Dog();
        for (int i=1; i<50; i++) {
            int randAnimalIndex = rand.nextInt(3);
            if (randAnimalIndex == 0) someAnimal = new Dog();
            else if (randAnimalIndex == 1) someAnimal = new Cat();
            else if (randAnimalIndex == 2) someAnimal = new Bigcat();
            someAnimal.PrintYourAngrySound();
        }
    }
}
```

Composition & Aggregation

Composition: A relationship where an object will not exist without the parent object. For example, it is unlikely that the "Nose" object will exist after the "Person" object is gone.

Aggregation: A relationship where multiple objects will likely continue to exist independent of each other. For example, we might have a "Household" object and a "Person" object. It would be very reasonable to expect our "Person" object to continue to exist even if our "Household" object was deleted.

Our First UML:

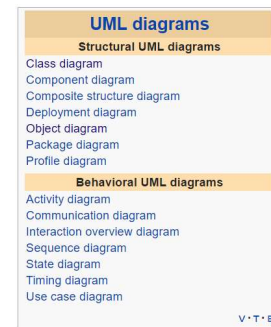


Unified Modeling Language [\[Link\]](#)

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system.

For our purposes we will limit our UML usage to diagrams where:

- Classes as boxes with three sections, the top of which specifies the name of the class, the middle of which specifies the data, and the bottom of which specifies the functions.
- Lines between the classes.
 - A line with an arrow / triangle pointing to the parent – inheritance
 - A line with a filled diamond next to the owner – composition
 - A line with an open diamond next to the owner – aggregation
 - A line with no decorations – just an association (a using kind of relationship)

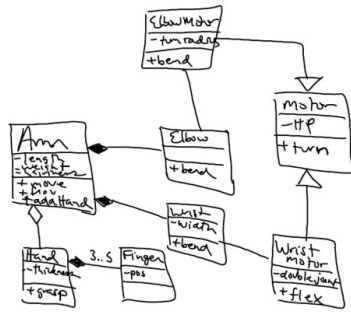


UML can describe many different things and become quite complex. There is some debate, particularly in the Agile development methodology circles, as to the benefits of this documentation outweigh the cost. Most developers agree that at least a basic understanding of UML diagrams aids greatly in the impromptu “chalkboard” design discussions that occur constantly during the design, implementation, and testing of a project.

UML helps us see the design without having to wade through lots of text to understand it.

UML Example: Robot Arm

UML:



Java:

```
// *****  
// Robot Arm  
  
class Arm {  
    private Elbow elb;  
    private Wrist wri;  
    private Hand hnd;  
    public Arm() {  
        elb = new Elbow();  
        wri = new Wrist();  
        hnd = null;  
    }  
    public void addHand(Hand h) {  
        hnd = h;  
    }  
}  
...  
  
Arm arm = new Arm();  
Hand hand = new Hand();  
arm.addHand(hand);
```

UML can describe many different things and become quite complex. There is some debate, particularly in the Agile development methodology circles, as to the benefits of this documentation outweigh the cost. Most developers agree that at least a basic understanding of UML diagrams aids greatly in the impromptu “chalkboard” design discussions that occur constantly during the design, implementation, and testing of a project.

UML helps us see the design without having to wade through lots of text to understand it.

Explain common design patterns

Define and demonstrate the common software patterns

Identify and describe characteristics of bad software: rigidity, immobility, fragility

Describe how object-oriented programming is fundamentally different

Justify the choice to use an object-oriented approach in developing software

Bonus Slides

- [Waterfall](#) vs [Iterative](#) vs [Agile](#)

Waterfall vs Iterative vs Agile

	Waterfall	Iterative	Agile
References	United States Department of Defense: DOD-STD-2167A (1985)	Rational Unified Process (RUP) Open Unified Process	Scrum Kanban Scaled Agile Framework (SAFe)
Priorities	Planning and predictability	Architecture, modeling, and efficiency through early detection & fixing of issues	Responsiveness to feedback, efficiency through engineering practices, early detection & fixing of issues
Principles	Execute phases sequentially: 1. Requirements 2. Analysis 3. Design 4. Coding 5. Testing 6. and Operations Define and commit to Scope, Cost, and Timeline “early” Implement strict Change Control	Develop and test iteratively Manage requirements Use components Model visually Verify quality Control changes	Develop, test, deploy, and release iteratively Capture lightweight near term requirements Empower teams Allow requirements to evolve but maintain fixed timelines Apply engineering practices and systems thinking (e.g. TDD) Integrate early user feedback into remaining plan Maintain a collaborative approach between all stakeholders

Waterfall vs Iterative vs Agile (continued)

	Waterfall	Iterative	Agile
Engineering Standards	None specified	Components and Modeling (UML) XP referenced periodically	XP DevOps
Optimization	Specialization & standardization based on skills, location, and/or technology	Architecture, modeling, and iterations	Fast customer feedback, small cross-functional teams, and working software
Planning Horizon Focus	Years – High Months – High Days – Low	Years – Medium Months – Medium Days – Medium	Years – Low Months – Medium Days – High
Scope and Requirements Management	Scope locked-in after Requirements Phase (approximately one-third through the project)	Requirements are stabilized during Elaboration Phase with smaller changes in later phases	Changes in requirements embraced utilizing lowest priority items as Scope buffer
Prevalent Issues	Timeline and cost overruns	Timeline and cost overruns	Reduced functionality release
Industry Success Rates	Low to Medium	Medium	Medium to High

Waterfall vs Iterative vs Agile (continued)

Optional

	Waterfall	Iterative	Agile
Delivery Ownership	Ownership changes from role to role by phase with Project Manager being responsible for overall delivery	Project Manager responsible for completing iteration	Team centric with team responsible for delivery of working feature
Resource Utilization	Role and skill centric with tasks often being restricted to an individual or group	Role and skill centric with tasks often being restricted to an individual or group	Team centric with team members switching roles regularly
Status Reporting	Work progress (% complete) relative to plan	Work progress plus product features delivered by Iteration during Implementation	Product features demonstrated by Sprint and Iteration
Customer Feedback	At the end of the project when the product is released	At the end of the project plus internally with each iteration	At the end of each release and iteration plus internally with each sprint
Retrospection	After each release	After each iteration or milestone	After each sprint
Quality Control	Detection & fixing primarily last phase of project	Early detection & fixing in each iteration and in last phase of project	Early detection & fixing in each sprint followed by stabilization