Effective Object Oriented Design

Robert Lowe

Division of Mathematics and Computer Science
Maryville College
Outline

1. Software Engineering Overview
2. Object Oriented Analysis and Design
3. Unified Modeling Language - UML
Outline

1. Software Engineering Overview
2. Object Oriented Analysis and Design
3. Unified Modeling Language - UML
Stages of Software Development

1. Requirement Gathering and Analysis
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

Reality Check: In the real world, the above is step 1. The software then enters a cycle until it is decommissioned!

Image Source: wikipedia.org
Software Development Lifecycle

Stages of Software Development

1. Requirement Gathering and Analysis
2. Design

Image Source: wikipedia.org
Software Development Lifecycle

Stages of Software Development

1. Requirement Gathering and Analysis
2. Design
3. Implementation

Image Source: wikipedia.org
Software Development Lifecycle

Stages of Software Development

1. Requirement Gathering and Analysis
2. Design
3. Implementation
4. Testing

Image Source: wikipedia.org
Software Development Lifecycle

Stages of Software Development

1. Requirement Gathering and Analysis
2. Design
3. Implementation
4. Testing
5. Deployment

Reality Check
In the real world, the above is step 1. The software then enters a cycle until it is decommissioned!

Image Source: wikipedia.org
Software Development Lifecycle

Stages of Software Development

1. Requirement Gathering and Analysis
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

Image Source: wikipedia.org

Reality Check
In the real world, the above is step 1. The software then enters a cycle until it is decommissioned!
Software Development Lifecycle

Stages of Software Development

1. Requirement Gathering and Analysis
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

Reality Check
In the real world, the above is step 1. The software then enters a cycle until it is decommissioned!

Image Source: wikipedia.org
Software engineering is any methodology by which requirements are gathered and software is designed.
Software engineering is any methodology by which requirements are gathered and software is designed.

Communication is the primary concern.
Software engineering is any methodology by which requirements are gathered and software is designed.

Communication is the primary concern.

SE is also used to determine success or failure in a software project.
Software Engineering Overview

- Software engineering is any methodology by which requirements are gathered and software is designed.
- Communication is the primary concern.
- SE is also used to determine success or failure in a software project.
- SE also provides a methodology of managing development resources and time.
Software engineering is any methodology by which requirements are gathered and software is designed.

Communication is the primary concern.

SE is also used to determine success or failure in a software project.

SE also provides a methodology of managing development resources and time.

There is no silver bullet that makes software easy to develop.
Software engineering is any methodology by which requirements are gathered and software is designed.

Communication is the primary concern.

SE is also used to determine success or failure in a software project.

SE also provides a methodology of managing development resources and time.

There is no silver bullet that makes software easy to develop.

Every year, someone attempts to promote a new "Silver Bullet".
Software engineering is any methodology by which requirements are gathered and software is designed.

Communication is the primary concern.

SE is also used to determine success or failure in a software project.

SE also provides a methodology of managing development resources and time.

There is no silver bullet that makes software easy to develop.

Every year, someone attempts to promote a new "Silver Bullet".

Ignore them.
A stakeholder is anyone involved with, or affected by, a project.
A stakeholder is anyone involved with, or affected by, a project.

Programmers
Communication with Stakeholders

- A stakeholder is anyone involved with, or affected by, a project.
- Programmers
  - High Technical Knowledge
Communication with Stakeholders

- A stakeholder is anyone involved with, or affected by, a project.
- Programmers
  - High Technical Knowledge
  - Low Business Acumen

Robert Lowe  Effective Object Oriented Design
Communication with Stakeholders

- A stakeholder is anyone involved with, or affected by, a project.

- Programmers
  - High Technical Knowledge
  - Low Business Acumen
  - Often Not a Domain Expert
A stakeholder is anyone involved with, or affected by, a project.

Programmers
- High Technical Knowledge
- Low Business Acumen
- Often Not a Domain Expert

Managers and Clients
Communication with Stakeholders

- A stakeholder is anyone involved with, or affected by, a project.

  - Programmers
    - High Technical Knowledge
    - Low Business Acumen
    - Often Not a Domain Expert

  - Managers and Clients
    - Low Technical Knowledge
Communication with Stakeholders

- A stakeholder is anyone involved with, or affected by, a project.

- Programmers
  - High Technical Knowledge
  - Low Business Acumen
  - Often Not a Domain Expert

- Managers and Clients
  - Low Technical Knowledge
  - High Business Acumen
A stakeholder is anyone involved with, or affected by, a project.

Programmers
- High Technical Knowledge
- Low Business Acumen
- Often Not a Domain Expert

Managers and Clients
- Low Technical Knowledge
- High Business Acumen
- Domain Experts
Communication with Stakeholders

- A stakeholder is anyone involved with, or affected by, a project.

- Programmers
  - High Technical Knowledge
  - Low Business Acumen
  - Often Not a Domain Expert

- Managers and Clients
  - Low Technical Knowledge
  - High Business Acumen
  - Domain Experts

- A big part of software engineering is putting these two worlds together!
The oldest software engineering method.
The oldest software engineering method.

It seems to make sense.
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
- When it works, it works really well.
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
- When it works, it works really well.
- When it fails, it is catastrophically expensive.
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
- When it works, it works really well.
- When it fails, it is catastrophically expensive.
- Process of Propositions and Sign-offs
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
- When it works, it works really well.
- When it fails, it is catastrophically expensive.
- Process of Propositions and Sign-offs
  1. Requirements
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
- When it works, it works really well.
- When it fails, it is catastrophically expensive.
- Process of Propositions and Sign-offs
  1. Requirements
  2. Design
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
- When it works, it works really well.
- When it fails, it is catastrophically expensive.
- Process of Propositions and Sign-offs
  1. Requirements
  2. Design
  3. Implementation
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
- When it works, it works really well.
- When it fails, it is catastrophically expensive.
- Process of Propositions and Sign-offs
  1. Requirements
  2. Design
  3. Implementation
  4. Verification
Software Engineering Model: Waterfall

- The oldest software engineering method.
- It seems to make sense.
- It rarely works well.
- When it works, it works really well.
- When it fails, it is catastrophically expensive.

Process of Propositions and Sign-offs

1. Requirements
2. Design
3. Implementation
4. Verification
5. Maintenance
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
**Agile Methodologies**

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)
- Examples of Agile Methodologies
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)
- Examples of Agile Methodologies
  - Extreme Programming
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)
- Examples of Agile Methodologies
  - Extreme Programming
  - Crystal Clear Methods
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)

Examples of Agile Methodologies

- Extreme Programming
- Crystal Clear Methods
- Feature Driven Development
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)
- Examples of Agile Methodologies
  - Extreme Programming
  - Crystal Clear Methods
  - Feature Driven Development
  - Adaptive Software Development
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)
- Examples of Agile Methodologies
  - Extreme Programming
  - Crystal Clear Methods
  - Feature Driven Development
  - Adaptive Software Development
  - Scrum
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)
- Examples of Agile Methodologies
  - Extreme Programming
  - Crystal Clear Methods
  - Feature Driven Development
  - Adaptive Software Development
  - Scrum
  - Test-Driven Development

Most agile methodologies assume OOP, and maybe some functional programming.
Agile Methodologies

- Waterfall is old and stodgy. Agile is young and hep!
- Agile methodologies are cyclic.
- The focus is on early prototyping, with heavy stakeholder involvement.
- Development continues until everyone is happy (or so unhappy they terminate the project.)
- Examples of Agile Methodologies
  - Extreme Programming
  - Crystal Clear Methods
  - Feature Driven Development
  - Adaptive Software Development
  - Scrum
  - Test-Driven Development
- Most agile methodologies assume OOP, and maybe some functional programming.
Outline

1. Software Engineering Overview
2. Object Oriented Analysis and Design
3. Unified Modeling Language - UML
Goal: Determine what the objects are and how they interact.
Object Oriented Analysis

**Goal:** Determine what the objects are and how they interact.

**Methodology**
Object Oriented Analysis

**Goal:** Determine what the objects are and how they interact.

**Methodology**
1. Gather requirements, get a basic understanding.
Goal: Determine what the objects are and how they interact.

Methodology
1. Gather requirements, get a basic understanding.
2. Create use cases to analyze user interactions.
Object Oriented Analysis

- **Goal:** Determine what the objects are and how they interact.

- **Methodology**
  1. Gather requirements, get a basic understanding.
  2. Create use cases to analyze user interactions.
  3. Tease out objects by studying use cases.
Goal: Determine what the objects are and how they interact.

Methodology:
1. Gather requirements, get a basic understanding.
2. Create use cases to analyze user interactions.
3. Tease out objects by studying use cases.
4. Sketch out how objects interact with each other.
Object Oriented Analysis

**Goal:** Determine what the objects are and how they interact.

**Methodology**
1. Gather requirements, get a basic understanding.
2. Create use cases to analyze user interactions.
3. Tease out objects by studying use cases.
4. Sketch out how objects interact with each other.
5. Create a rudimentary object/class layout.
**Goal**: Create a detailed technical design which fully specifies classes, attributes, and methods.
**Goal:** Create a detailed technical design which fully specifies classes, attributes, and methods.

**Methodology**
Object Oriented Design

Goal: Create a detailed technical design which fully specifies classes, attributes, and methods.

Methodology
1. Group objects together to identify classes.
Object Oriented Design

**Goal:** Create a detailed technical design which fully specifies classes, attributes, and methods.

**Methodology**

1. Group objects together to identify classes.
2. Use use cases and object descriptions to create attributes.
**Goal:** Create a detailed technical design which fully specifies classes, attributes, and methods.

**Methodology**

1. Group objects together to identify classes.
2. Use use cases and object descriptions to create attributes.
3. Use object interactions and use cases to create object methods.
Objective: Create a detailed technical design which fully specifies classes, attributes, and methods.

Methodology:
1. Group objects together to identify classes.
2. Use use cases and object descriptions to create attributes.
3. Use object interactions and use cases to create object methods.
4. Analyze classes, looking for commonalities.
Goal: Create a detailed technical design which fully specifies classes, attributes, and methods.

Methodology

1. Group objects together to identify classes.
2. Use use cases and object descriptions to create attributes.
3. Use object interactions and use cases to create object methods.
4. Analyze classes, looking for commonalities.
5. Establish abstract classes and inheritance relationships.
Building Flexible Designs

- The real key to successful OOADP is to make flexible and reusable object.
The real key to successful OOADP is to make flexible and reusable object.

Design cohesive objects! Each object should be a thing unto itself.
The real key to successful OOADP is to make flexible and reusable object.

Design cohesive objects! Each object should be a thing unto itself.

Objects which do too much should likely be split apart.
Building Flexible Designs

- The real key to successful OOADP is to make flexible and reusable object.
- Design cohesive objects! Each object should be a thing unto itself.
- Objects which do too much should likely be split apart.
- If it’s not a noun, it’s not an object!
Building Flexible Designs

- The real key to successful OOADP is to make flexible and reusable object.
- Design cohesive objects! Each object should be a thing unto itself.
- Objects which do too much should likely be split apart.
- If it’s not a noun, it’s not an object!
- Create and abide by your object’s method interfaces.
Building Flexible Designs

- The real key to successful OOADP is to make flexible and reusable object.
- Design cohesive objects! Each object should be a thing unto itself.
- Objects which do too much should likely be split apart.
- If it’s not a noun, it’s not an object!
- Create and abide by your object’s method interfaces.
- Play by your own rules. No object should be tightly coupled with non-public behavior of another object.
The real key to successful OOADP is to make flexible and reusable object.

Design cohesive objects! Each object should be a thing unto itself.

Objects which do too much should likely be split apart.

If it’s not a noun, it’s not an object!

Create and abide by your object’s method interfaces.

Play by your own rules. No object should be tightly coupled with non-public behavior of another object.

Remember, agile implies changes, write general objects which can cope with change!
Outline

1. Software Engineering Overview
2. Object Oriented Analysis and Design
3. Unified Modeling Language - UML
UML Overview

- A graphical design language.

Diagram:
- Programmer
  - Gathers Requirements
  - Analyzes Objects
  - Designs Classes
UML Overview

- A graphical design language.
- Programming Language Agnostic.
UML Overview

- A graphical design language.
- Programming Language Agnostic.
- Comprised several types of diagrams.
UML Overview

- A graphical design language.
- Programming Language Agnostic.
- Comprised several types of diagrams.
  - Use Case Diagram
UML Overview

- A graphical design language.
- Programming Language Agnostic.
- Comprised several types of diagrams.
  - Use Case Diagram
  - Object Diagram

Gathers Requirements

Programmer

Analyzes Objects

Designs Classes
UML Overview

- A graphical design language.
- Programming Language Agnostic.
- Comprised several types of diagrams.
  - Use Case Diagram
  - Object Diagram
  - Sequence Diagram
UML Overview

- A graphical design language.
- Programming Language Agnostic.
- Comprised several types of diagrams.
  - Use Case Diagram
  - Object Diagram
  - Sequence Diagram
  - Class Diagram
UML Overview

- A graphical design language.
- Programming Language Agnostic.
- Comprised several types of diagrams.
  - Use Case Diagram
  - Object Diagram
  - Sequence Diagram
  - Class Diagram
- Used to Design, Communicate, and Clarify Software Systems
Use Case Diagrams

- A Use Case diagram outlines the ways in which a system will be used.
A Use Case diagram outlines the ways in which a system will be used.

Each class of actor performs a series of actions.
Use Case Diagrams

- A Use Case diagram outlines the ways in which a system will be used.
- Each class of actor performs a series of actions.
- Actors are represented as a stick figure.
Use Case Diagrams

- A Use Case diagram outlines the ways in which a system will be used.
- Each class of actor performs a series of actions.
- Actors are represented as a stick figure.
- Performance of tasks is represented as a line.
Use Case Diagrams

- A Use Case diagram outlines the ways in which a system will be used.
- Each class of actor performs a series of actions.
- Actors are represented as a stick figure.
- Performance of tasks is represented as a line.
- Often differentiates different types of users.
A class diagram shows specification of the program’s classes.
Class Diagrams

- A class diagram shows specification of the program’s classes.
- Inheritance Relationships

```
Widget

Shape
- __complete : bool
+ Shape()
+ handleEvent( e : Event ) : void
+ addPoint( _x : int, _y : int ) : void «pure virtual»
+ isComplete() : bool
```
A class diagram shows specification of the program’s classes.

Inheritance Relationships

Class Attributes
A class diagram shows specification of the program’s classes.

- Inheritance Relationships
- Class Attributes
- Class Methods
Analysis begins with conversations and study of the problem at hand.
Analysis begins with conversations and study of the problem at hand.

The first diagram, or set of diagrams, to draw is the Use Case diagram.
Analysis begins with conversations and study of the problem at hand.

The first diagram, or set of diagrams, to draw is the Use Case diagram.

Examine the use case diagram, review them with clients, and add/remove use cases as needed.
Analysis begins with conversations and study of the problem at hand.

The first diagram, or set of diagrams, to draw is the Use Case diagram.

Examine the use case diagram, review them with clients, and add/remove use cases as needed.

Use the use case diagram to draw a rudimentary class diagram, with class names and relationships only.
Using your use case diagram, and your rudimentary class diagram, work out necessary methods and attributes for your classes.
Using your use case diagram, and your rudimentary class diagram, work out necessary methods and attributes for your classes.

Add methods and attributes to your class diagram.
Using your use case diagram, and your rudimentary class diagram, work out necessary methods and attributes for your classes.

Add methods and attributes to your class diagram.

Identify classes which should have super classes, and build inheritance relationships.
Using your use case diagram, and your rudimentary class diagram, work out necessary methods and attributes for your classes.

Add methods and attributes to your class diagram.

Identify classes which should have super classes, and build inheritance relationships.

Split classes that don’t make sense as cohesive objects.
Using your use case diagram, and your rudimentary class diagram, work out necessary methods and attributes for your classes.

Add methods and attributes to your class diagram.

Identify classes which should have super classes, and build inheritance relationships.

Split classes that don’t make sense as cohesive objects.

Keep building the class diagram until it fully models your program!
Using your use case diagram, and your rudimentary class diagram, work out necessary methods and attributes for your classes.

Add methods and attributes to your class diagram.

Identify classes which should have super classes, and build inheritance relationships.

Split classes that don’t make sense as cohesive objects.

Keep building the class diagram until it fully models your program!

Use the class diagram as a road map when you move on to implementation.
Case Study: Object Oriented Pong

Let’s perform OOAD on pong, using UML!