Recap: Software Engineering

What is Software Engineering?
The complete process of specifying, designing, developing, analyzing, deploying, and maintaining a software system.

Why is it important?
- Software is everywhere and complex.
- Software defects are expensive (and annoying).

Goals
- Decompose a complex engineering problem.
- Organize processes and effort.
- Improve software reliability.
- Improve developer productivity.

Today
- Modeling and abstraction
- Software architecture vs. software design
- UML crash course
Software development: the high-level problem

One solution: “Here happens a miracle”

Software development: the high-level problem

Another solution: Modeling the architecture and design

What is modeling?

Building an abstract representation of reality
- Ignoring (insignificant) details.
- Level of abstraction depends on viewpoint and purpose:
  - Communication
  - Verification
  - Code generation
- Focusing on the most important aspects/properties.

Is abstraction == simplification?
Different levels of abstraction

Source code

Example: Linux Kernel
- 16 million Lines of Code!
- What does the code do?
- Are there dependencies?
- Are there different layers?

Software architecture vs. design

Architecture (what components are developed?)
- Considers the system as a whole:
  - High-level view of the overall system.
  - What components exist?
  - What type of storage, database, communication, etc?

Design (how are the components developed?)
- Considers individual components:
  - Data representation
  - Interfaces, Class hierarchies
  - ...

Architecture vs. design

Development process

Specification

Architecture

Level of abstraction

Design

Source code

What’s the difference?

Call graph

Layer diagram
A first example

Goal: group and count CS520 grades.

Architecture or design pattern?

Software architecture: Pipe and Filter

The architecture doesn’t specify the design or implementation details of the individual components (filters)!

Software architecture: Client-server / n-tier

Simplifies reusability, exchangeability, and distribution.
Software architecture: **Model View Controller**

- **Client**
  - sees
  - uses
- **Controller**
  - updates
  - manipulates
- **Model**
  - Separates data representation (Model), visualization (View), and client interaction (Controller)

---

Model View Controller: example

**Simple weather station**

- **Current**
  - 25° F
  - -4° C
- **30 day history**
  - min: 20° F
  - max: 35° F
- **Reset**
- **Reset history button**

---

Model View Controller: example

**Simple weather station**

- **Current**
  - 25° F
  - -4° C
- **30 day history**
  - min: 20° F
  - max: 35° F
- **Reset**
- **Reset history button**

---

Summary: Software architecture vs. design

**Architecture and design goals**
- Lower complexity: separation of concerns, well defined interfaces
- Simplify communication
- Allow effort estimation and progress monitoring

---

Presentation layer

Business logic layer

Data access layer
UML crash course

The main questions
- What is UML?
- Is it useful, why bother?
- When to (not) use UML?

What is UML?
- Unified Modeling Language.
- Developed in the mid 90’s, improved since.
- Standardized notation for modeling OO systems.
- A collection of diagrams for different viewpoints:
  - Use case diagrams
  - Component diagrams
  - Class and Object diagrams
  - Sequence diagrams
  - Statechart diagrams
  - ...
Are UML diagrams useful?

Communication
- Forward design (before coding)
  - Brainstorm ideas (on whiteboard or paper).
  - Draft and iterate over software design.
- Backward design (after coding)
  - Obtain diagram from source code.

Documentation
- Generating source code from diagrams is challenging.
- Code generation may be useful for skeletons.

In this class, we will use UML class diagrams mainly for visualization and discussion purposes.

Classes vs. objects

Class
- Grouping of similar objects.
  - Student
  - Car
- Abstraction of common properties and behavior.
  - Student: Name and Student ID
  - Car: Make and Model

Object
- Come from the real world.
- Instance of a class
  - Student: Juan (4711), Jane (4712), ...
  - Car: Audi A6, Honda Civic, Tesla S, ...

UML class diagram: basic notation

MyClass
UML class diagram: basic notation

Name

MyClass
- attr1 : type
# attr2 : type
+ attr3 : type

Methods
- bar(a:type) : ret_type
+ foo() : ret_type

Visibility
- private
# package-private
+ protected
+ public

Attributes
- attr1 : type
# attr2 : type
+ attr3 : type

Static attributes or methods are underlined

UML class diagram: basic notation

Name

MyClass
- attr1 : type
# attr2 : type
+ attr3 : type

Methods
- bar(a:type) : ret_type
+ foo() : ret_type

Visibility
- private
# package-private
+ protected
+ public

Attributes
- attr1 : type
# attr2 : type
+ attr3 : type

So why bother with UML when you have code?
Classes, abstract classes, and interfaces

```java
public abstract class MyAbstractClass {
    public abstract void op();
    public int op2();
}

public interface MyInterface {
    public void op();
    public int op2();
}

public class MyClass {
    public void op() {
        ...
    }
    public int op2() {
        ...
    }
}
```

Level of detail in a given class or interface may vary and depends on context and purpose.

UML class diagram: Inheritance

```
public class SubClass extends SuperClass implements AnInterface
```

UML class diagram: Aggregation and Composition

- **Aggregation**
  - Part has-a relationship
  - Whole
  - Existence of Part does not depend on the existence of Whole.
  - Lifetime of Part does not depend on Whole.
  - No single instance of whole is the unique owner of Part (might be shared with other instances of Whole).

- **Composition**
  - Part has-a relationship
  - Whole
  - Part cannot exist without Whole.
  - Lifetime of Part depends on Whole.
  - One instance of Whole is the single owner of Part.
Aggregation or Composition?

Room
??
Building
??
Customer
??
Bank

Composition
Room
??
Customer
??
Building
??
Bank

What about class and students or body and body parts?

UML class diagram: multiplicity

A
1
-1
B
Each A is associated with exactly one B
Each B is associated with exactly one A

A
1..2
-1
B
Each A is associated with any number of Bs
Each B is associated with exactly one or two As

UML class diagram: navigability

A
Navigability: not specified
B
A
Navigability: unidirectional
“can reach B from A”
B
A
Navigability: bidirectional
B
Summary: UML

- Unified notation for modeling OO systems.
- Allows different levels of abstraction.
- Suitable for design discussions and documentation.
- Generating code from diagrams is challenging.