
[CS3704] Software Engineering

Dr. Chris Brown

Virginia Tech

10/25/2023

Announcements

- **Discussion Presentation on Design
Friday (10/27) in class**

Low-Level Design II

Software Design Quality
Designing for Reuse
Design in Software Engineering

Learning Outcomes

By the end of the course, students should be able to:

- **Understand software engineering processes, methods, and tools used in the software development life cycle (SDLC)**
- Use techniques and processes to create and analyze requirements for an application
- **Use techniques and processes to design a software system**
- Identify processes, methods, and tools related to phases of the SDLC
- Explain the differences between software engineering processes
- Discuss research questions and current topics related to software engineering
- Create and communicate about the requirements and design of a software application

Goal: decide the structure of the software and the hardware configurations that support it.

- The *how* of the project
- How individual classes and software components work together in the software system.
 - Programs can have 1000s of classes/methods
- *Software Artifacts:* design documents, class diagrams (i.e. UML)

- The process of making decisions about HOW to implement software solutions to meet requirements.
- Encompasses the set of concepts, principles, and practices that lead to the development of high-quality systems.

Design Practices

- **High-level: Architecture design**
 - Define major components and their relationship
- **Low-level: Detailed design**
 - Decide classes, interfaces, and implementation algorithms for each component

Low-Level Design

- **Goal:** To decompose subsystems into modules
- Two approaches:
 1. Procedural:
 - system is decomposed into functional modules which accept input data and transform it to output data
 - achieves mostly procedural abstractions
 2. Object-oriented
 - system is decomposed into a set of communicating objects
 - achieves both procedural + data abstractions

Warm-Up

Discuss:

- Do you design software when you write a program?
- What is the difference between the creational, structural, and behavioral low-level design pattern families?

Design Pattern Families

Creational

Concerned with the process of object creation

- Increases flexibility and reuse of code

Structural

Deal with the composition of classes or objects

- Organizing different classes and modules to form larger structures or add new functionality

Behavioral

Characterize the ways in which classes or objects interact and distribute responsibility

- Algorithms and assignment of responsibilities between objects

How Do Developers Design Software?

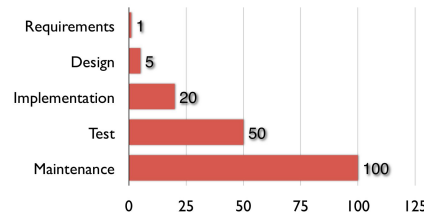
- **Code**
 - Design-while-coding 🥵
- **Iterative and evolutionary design**
 - Diagrams and modeling
 - UML
 - Class diagrams
 - Sequence diagrams
- **Reuse or modify existing design models**
 - High-level: Architectural patterns
 - Low-level: Design patterns

Software Design

“The most common miracle of software engineering is the transition from analysis to design and design to code.” [Due]

- Design is important because it allows a software team to assess the *quality* of the software **before it is implemented**—at a time when errors, omissions, or inconsistencies are easy and inexpensive to correct.

Reminder:



Design Quality

How do you assess the quality of design?

- Design must implement all of the explicit requirements and accommodate all implicit requirements
- Design must be readable, understandable, etc. for those who generate, test, and support the software.
- Design should provide a complete picture of the software (data, functionality, and behavior)

– Quality Guidelines [McGlaughlin]

Design Quality Guidelines

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics, and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics

Design Quality Guidelines (cont.)

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirement analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Design with Reuse

- Reuse-based software engineering is an approach which tries to maximize the reuse of existing software.
[Sommerville]
- Involves designing software around existing examples of good design and working components, making use of these when available.
- Components designed for reuse should be independent, reflect stable abstractions, provide access through interface operations, and *not* handle exceptions.
 - Ex) external or internal API libraries, commercial off the shelf (COTS) software, configuration and utility methods, etc.

Design with Reuse (cont.)

- **Application reuse:** application may be reused by incorporating it into other systems or developing application families that may run on different platforms specialized to the needs of customers.
- **Component reuse:** components of applications (subsystem to single object) may be reused
- **Function reuse:** single functions may be reused

Design with Reuse (cont.)

Benefits:

- Increased reliability
- Reduced risk
- Effective use of developers
- Standards compliance (i.e. data privacy)
- Accelerated development
 - Compared to adding new components

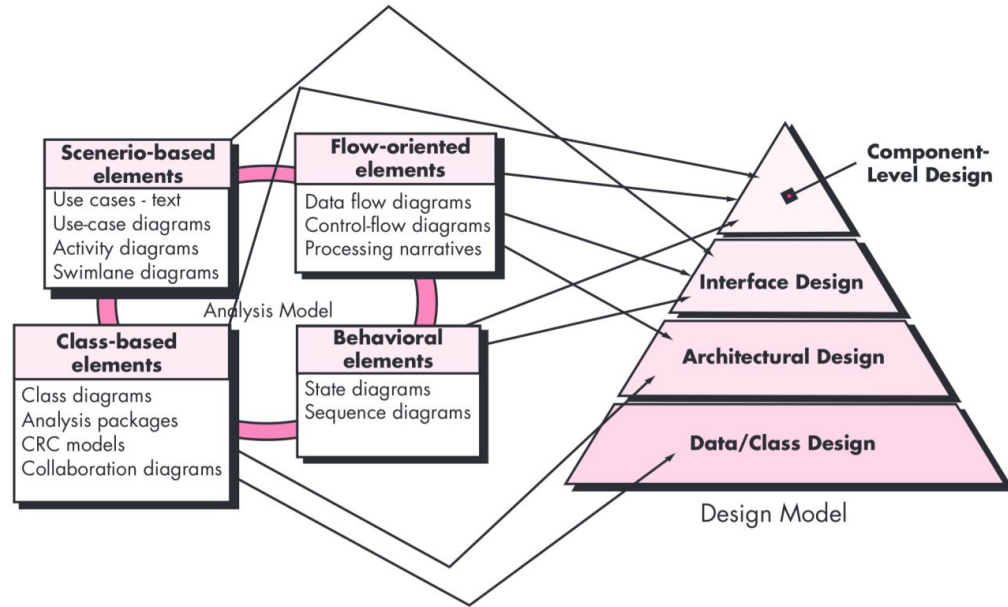
Problems:

- Increased maintenance for older components
- Lack of tool support
- Not-invented-here syndrome (API libraries)
- Finding and adapting reusable components

Design in SE Contexts

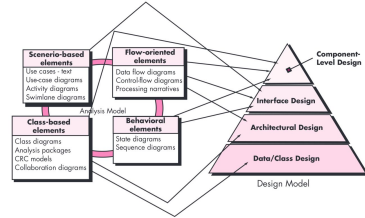
- Software design is applied regardless of the software process model used.
 - Begins once software requirements have been analyzed and modeled
- Design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing)
 - “During design, you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained” [Pressman]

Translating Req. Models to Design



[Pressman]

Design Models



Data/Class Design:

- Defines the data and attributes required to implement the software
- Ex) ER Diagrams, Data structures

Architectural (High-Level) Design:

- Defines the relationship between major structural elements of the software
- Ex) High-level design patterns

Interface Design:

- Defines how software communicates with systems and humans
- Ex) Prototyping, Wireframing, Storyboarding,...

Component (Low-Level) Design:

- Transforms structural elements into component-based flow and behavior
- Ex) Low-level design patterns

How is UML Really Used?

“UML has been described by some as ‘the lingua franca of software engineering’. Evidence from industry does not necessarily support such endorsements. How exactly is UML being used in industry – if it is? This paper presents a corpus of interviews with 50 professional software engineers in 50 companies and identifies 5 patterns of UML use.” [Petre]

<u>NONE!</u>	70%
SELECTIVE	22%
AUTOMATIC CODE GEN	6%
RETROFIT	2%
WHOLE	0%

Of those that reported using it...

TABLE II. ELEMENTS OF UML USED BY THE 11 ‘SELECTIVE’ USERS.

UML diagrams	Number of users	Reported to be used for...
Class diagrams	7	structure, conceptual models, concept analysis of domain, architecture, interfaces
Sequence diagrams	6	requirements elicitation, eliciting behaviors, instantiation history
Activity diagrams	6	modeling concurrency, eliciting useful behaviors, ordering processes
State machine diagrams	3	
Use case diagrams	1	represent requirements

Guidelines

- Spend significant time doing **dynamic modeling**, not just static modeling
 - Do static modeling after dynamic modeling
- Apply **responsibility-driven design** principles to dynamic modeling

Static and Dynamic Modeling

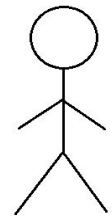
- Dynamic models
 - help design the logic or behaviors of the code
 - UML interaction diagrams
 - Ex) sequence diagrams
- Static models
 - help design the definition of packages, class names, attributes, and method signatures
 - Ex) UML class diagrams

Revisiting Sequence Diagrams

- A notation to illustrate actor interactions and operations initiated by them
- Only the interaction between users and the system is modeled in system sequence diagram
 - Allows for more advanced computational interaction (*i.e.*, conditional statements, loops, etc.)

UML Sequence Diagram (cont.)


- Graphical Depiction



Actor

A rectangular box representing an object in a UML sequence diagram.

Object

A tall, narrow rectangular box representing an activation bar in a UML sequence diagram.

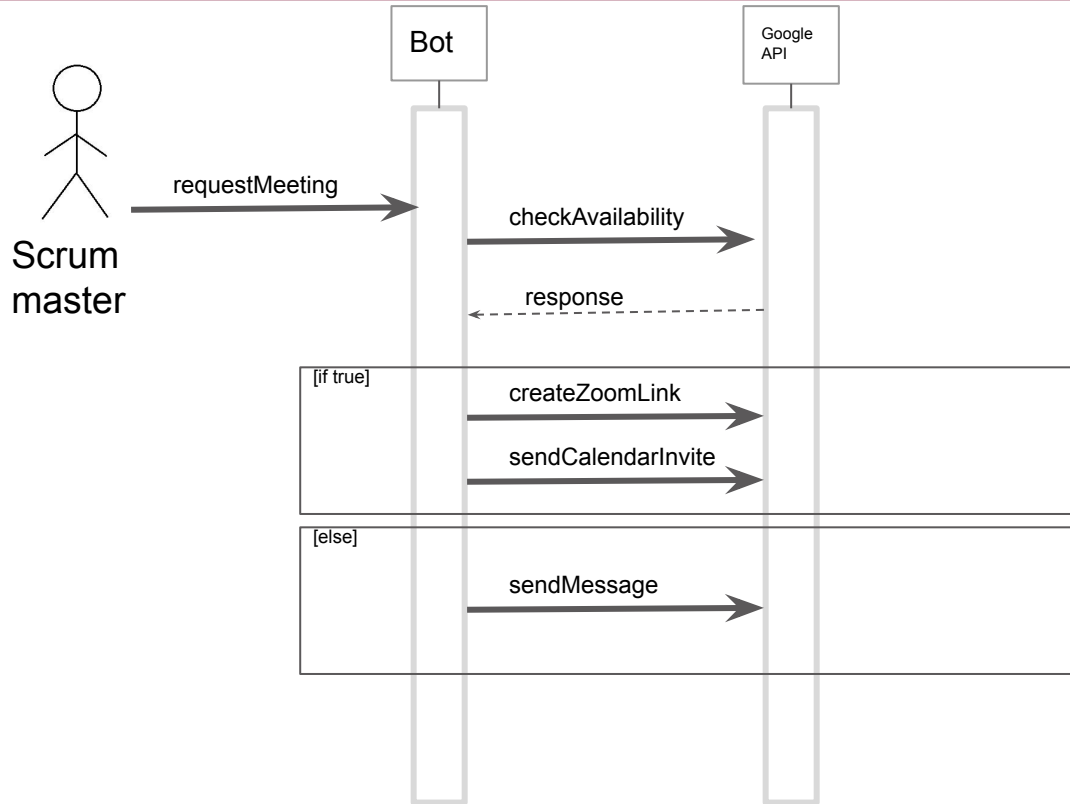
Activation box:
represents time
needed to for object to
complete a task

[condition]

message/data →

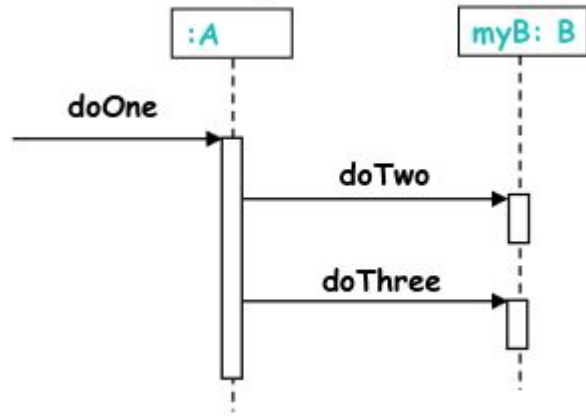
← Return message

Example: Stand-Up Bot



UML Interaction Diagrams

- Illustrate how objects interact with each other.



What does this look like in code?

```
public class A {
    private B myB = new B();
    public void doOne() {
        myB.doTwo();
        myB.doThree();
    }
}
```

Responsibility-Driven Design (RDD)

- Identify responsibilities (obligation or behavior of an object) and assign them to classes and objects
 - Primarily for object-oriented programming
- Think about how to assign responsibilities to collaborating objects
- Think about following questions:
 - What are the responsibilities of an object?
 - Who does it collaborate with?
 - What design patterns should be applied?
- Avoid “forcing” design patterns or over-engineering

RDD: Responsibilities

- Obligations or behavior of an object in terms of its role
- Responsibilities for *doing*
- Responsibilities for *knowing*

Responsibilities for Doing

Self-behaviors and collaborations or interactions with others

- Doing something itself, such as creating an object or doing a calculation
- Initiating action in other objects
- Controlling and coordinating activities in other objects
 - *Examples: create an object, perform a calculation, invoke an operation on other object, etc.*
- The transition of responsibilities into classes and methods is influenced by the granularity of the responsibility
 - Big responsibilities may involve hundreds of classes and methods
 - Little responsibilities may take one method

Responsibilities for Knowing

Self-data and relevant objects or data

- Knowing about private encapsulated data
- Knowing about related objects
- Knowing about things it can derive or calculate
 - *Examples: attributes, data involved in calculations, parameters when invoking operations, etc.*
- The attributes and associations illustrated by domain objects in a domain model often inspire the responsibilities

Responsibility Principles

- An approach to guide assignment of responsibilities [design and implementation]
 - **Creator:** who creates an object?
 - **Information expert:** Who has the information to fulfill responsibility?
 - **Low coupling:** How to reduce the impact of change?
 - **Controller:** What object receives and controls system operations?
 - **High cohesion:** How to keep an object focused and manageable?

Design Pattern Usage in SE


Do software engineers actually use design patterns?

- Not in terms of modeling (i.e. class diagrams)
- But yes
 - Design patterns promote reuse in software
 - Apply repeatable solutions to common problems that can be transformed directly into code
 - Make code more maintainable, easier to understand (for future you and others),...

Rest of Class This Semester

- Continue overview of the software development lifecycle phases...

Exam Details

- Online exam
 - Canvas Quiz
 - Accessible online Friday (11/3) 8:00am - 11:59pm
 - 1 hour to complete
 - *i.e., Start before 10:59pm Friday night!*
 - Open-book, notes, lecture slides, etc.
 - You may **NOT** work with others 
 - Exam Review in class next week (11/1)
 - All content except presentations and guest lecture
-

Exam Details (cont.)

Three sections:

1. Key Terms

- a. Matching
- b. Multiple choice (multi-select, T/F,...)

2. Scenarios

- a. Open-ended questions
 - i. You do not need to write more than **2-3 sentences max**
 - ii. Be specific (generic answers like “requirements” are **✗**)
 - iii. Do not repeat concepts for multiple questions (i.e. “scrum” can only be used once)

3. Application

- a. Mix (Multiple choice, short answer, open-ended,...)
 - b. Analyze code and be able to apply various concepts discussed in class.
-

Rest of Class Today

Design Pattern Workshop

- Form a small group
 - up to four, *must work in a group* (not your project group)
- Pick **one** pattern discussed and **one *not*** discussed in the last class
 - 10/23: Singleton, Builder, Factory, Abstract Factory, Adapter, Facade, Chain of Responsibility, State, Strategy
- Create a presentation to explain these design patterns to the class
 - You will not actually present to the class for this activity
 - Presentation should include the pattern, pattern family, definition, real-life and code examples, and any online resources used.
 - Turn in on Canvas with your group member names and PID on title slide.
 - Graded based on attendance* and effort
 - *Everyone must submit the slides on Canvas (by 2:20pm)*

Next Time...

- **Design Discussion Presentations Friday (10/27) in class**
- Code metrics on Monday (3/30)
- Exam Review (11/1)
- Exam (11/3)

References

- RS Pressman. *“Software engineering: a practitioner's approach”*.
- <https://refactoring.guru/design-patterns>
- Ian Sommerville. *“Software Engineering”*.
- Emil Vassev, Joey Paquet. *“Java Design Patterns”*. 2006-2012.
Concordia University
- Na Meng and Barbara Ryder
- Chris Parnin
- Sarah Heckman