# 17-723: Designing Large-scale Software Systems

**Course Review**

**April 22, 2024**

Carnegie Mellon University

# This Lecture

- Recall important design concepts and principles
- Describe the connections between the topics

# Learning Objectives

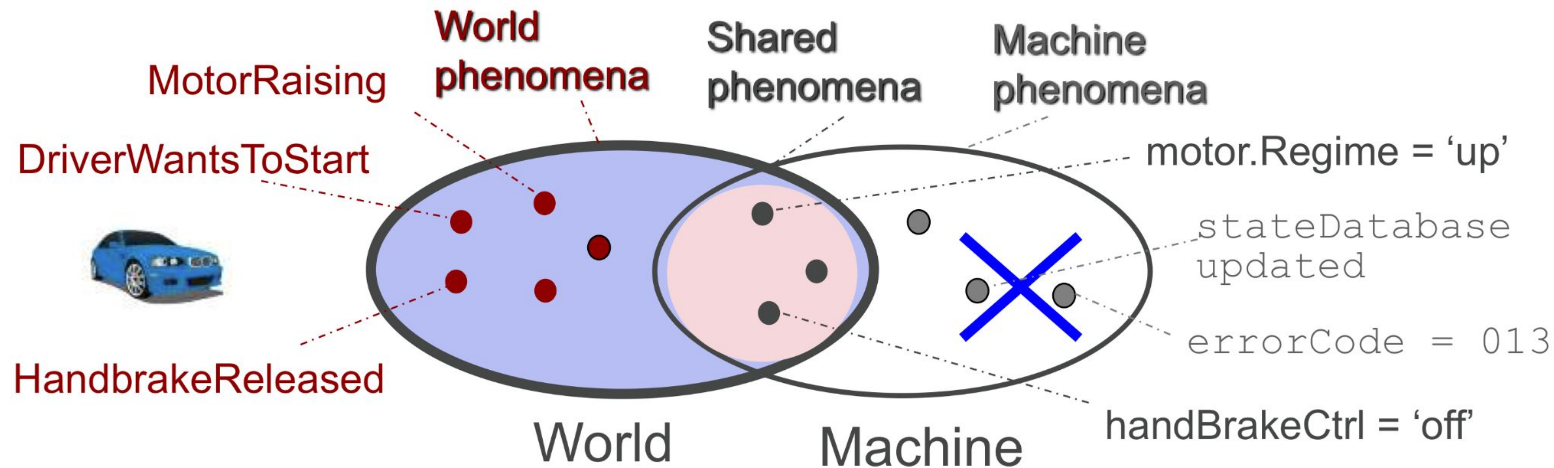
- Describe, recognise, and apply principles for: Design for reuse, design with reuse, design for change, design for robustness, design for testability, and design for scale
- Explain how to adapt a software design process to fit different domains, such as robotics, web apps, mobile apps, and medical systems
- Identify, describe, and prioritize relevant requirements for a given design problem
- Generate viable design solutions that appropriately satisfy the trade-offs between given requirements
- Apply appropriate abstractions & modeling techniques to communicate and document design solutions
- Evaluate design solutions based on their satisfaction of common design principles and trade-offs between different quality attributes

# Course Roadmap

**Foundational concepts & techniques for design**
Domain & design modeling, quality attributes & trade-offs, design space, generating design alternatives, design review, design processes

**Designing for quality attributes**
Design for change, interoperability, testability, reuse, scalability, robustness, security, usability, AI, ethics

# Problem vs. Solution Space




- Software (**solution space**) is one part of the system, and have limited control over the rest of the world (**problem space**)
- **Domain assumptions** are just as critical in achieving requirements
  - If you ignore/misunderstand these, your system may fail or do poorly (no matter how perfect your software is)
- Identify and document these assumptions as early as possible

# Quality Attributes (QAs)

- Functionality is just one aspect of software
- QAs are keys to making your product successful
- QAs should be specified in a way that is **measurable** and describing a **scenario** that your system handles
- QAs often conflict with each other! Consider **trade-offs** and prioritize for ones that are most important

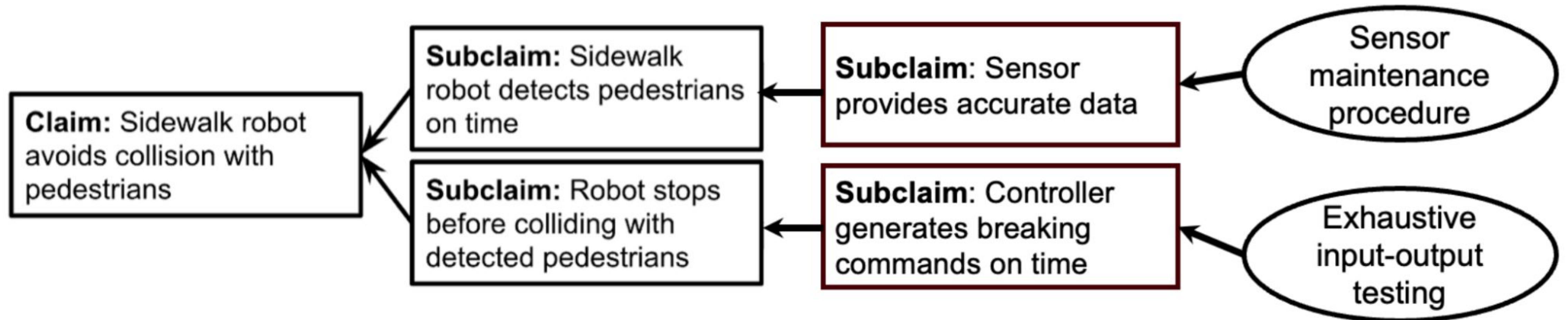
# Generating Design Alternatives

| Style | Constituent parts | | Control issues | | | Data issues | | | | Control/data interaction | | Type of reasoning |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Components | Connectors | Topo-logy | Synch-ronicity | Binding time | Topo-logy | Contin-uity | Mode | Binding time | Isomorphic shapes | Flow directions | |
| **Data-centered repository styles:** Styles dominated by a complex central data store, manipulated by independent computations | | | | | | | | | | | | Data integrity |
| Transactional data-base [Be90, Sp87] | memory, computations | trans. streams (queries) | star | asynch, opp | w | star | spor lvol | shared, passed | w | possibly | if isomorph-ic, opposite | ACID[5] properties |
| •Client/server | managers, computations | transaction opns with history[3] | star | asynch. | w, c, r | star | spor lvol | passed | w, c, r | yes | opposite | |
| Blackboard [Ni86] | memory, computations | direct access | star | asynch, opp | w | star | spor lvol | shared, mcast | w | no | n/a | convergence |
| Modern compiler [SG96] | memory, computations | procedure call | star | seq | w | star | spor lvol | shared | w | no | n/a | invariants on parse tree |

- Avoid sticking to the first design option that you think of (anchoring)
- Think of **multiple design options**!
  - Even if you are sure that you have enough, try to think of more
- Discuss the options with other team members; this may generate additional options
- Keep a catalog of design patterns, but do not overuse them

*A field guide to boxology: Preliminary classification of architectural styles for software systems* (Shaw & Clements, 1997)

# Arguing for Design




- You must be able to provide a sound **argument** (with **evidence**) that your design achieves intended functionality & QAs
  - If you can't come up with an argument, how do you know it works?
- **Assurance case** is one way to structure your arguments
- Apply **adversarial thinking** to find weaknesses and improve your argument

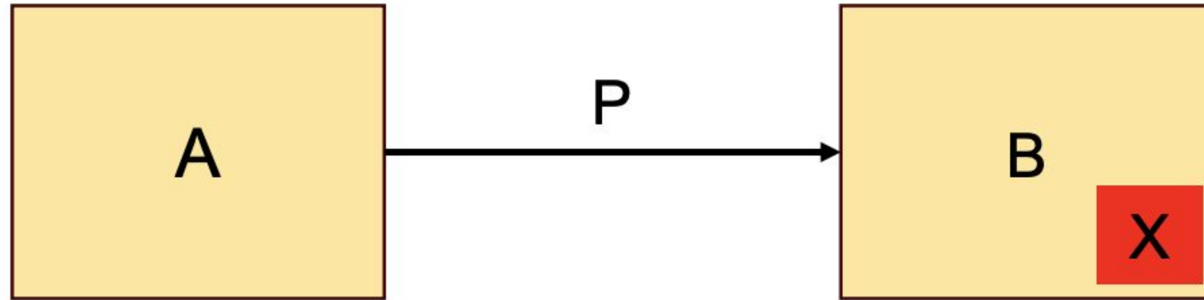
Carnegie Mellon University

# Course Roadmap

**Foundational concepts & techniques for design**
Domain & design modeling, quality attributes & trade-offs, design space, generating design alternatives, design review, design processes

**Designing for quality attributes**
Design for change, interoperability, testability, reuse, scalability, robustness, security, usability, AI, ethics

# Design for Changeability

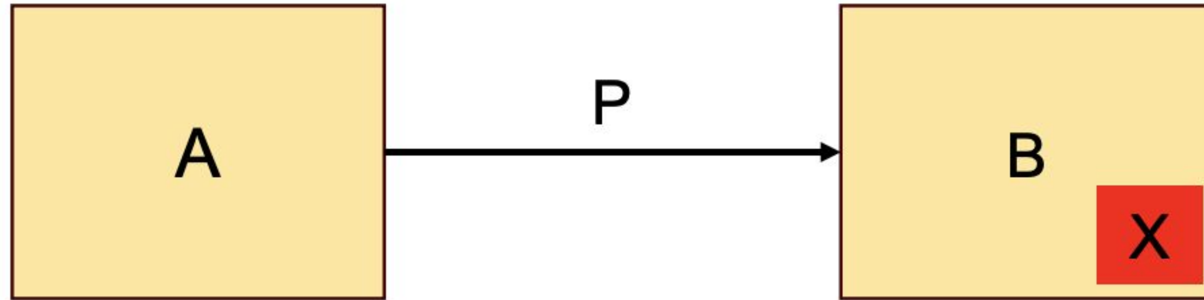



P: Public interface over B
X: Secret hidden in B
**Design task**
P should be designed so that changing X does not affect it

**Benefit**
Changing X does not affect A!

- **Changeability**: The amount of effort involved in making a particular change to a system
- **Key concept: Dependency** between components
  - Higher the degree of dependency, more you will need to change
- **Information hiding**: ??

# Design for Changeability



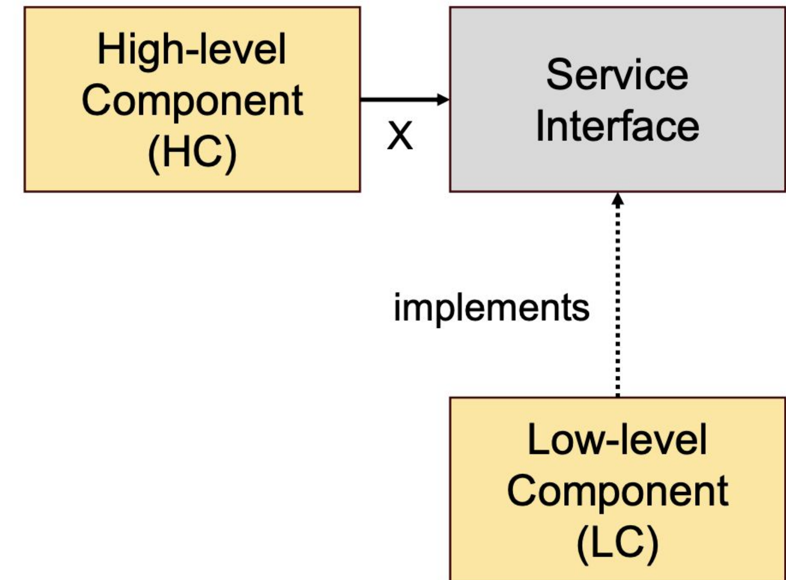

P: Public interface over B
X: Secret hidden in B
**Design task**
P should be designed so that changing X does not affect it

**Benefit**
Changing X does not affect A!

- **Changeability**: The amount of effort involved in making a particular change to a system
- **Key concept: Dependency** between components
  - Higher the degree of dependency, more you will need to change
- **Information hiding**: Hide secrets that are likely to change behind a component interface

# Design for Changeability: SOLID Principles

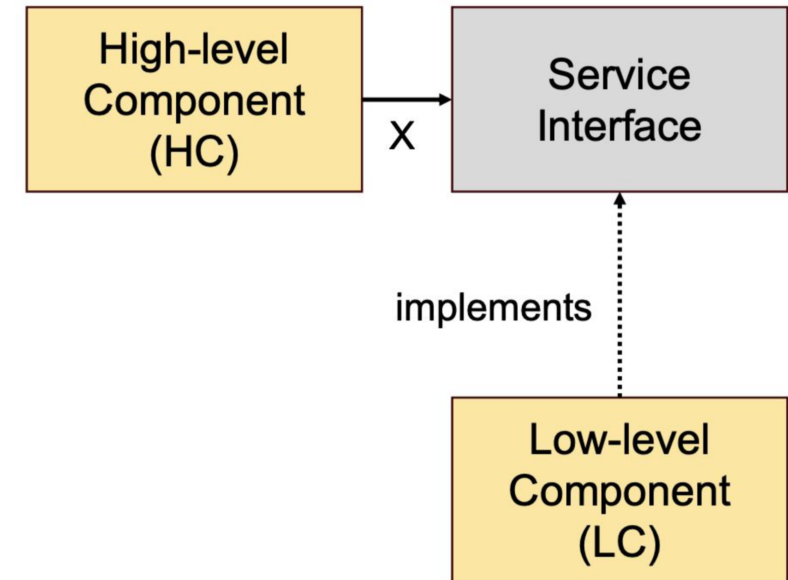
- **Single responsibility: ??**
- **Interface segregation: ??**
- **Dependency inversion principle: ??**




**Remember**: Changeability also adds **complexity** & **costs** to your system!

# Design for Changeability: SOLID Principles

- **Single responsibility:** Each component should be responsible for fulfilling a single purpose
- **Interface segregation:** An interface should not force its clients to depend on unnecessary details
- **Dependency inversion principle:** "High-level", application-logic component should not depend on "low-level", general-purpose components




**Remember**: Changeability also adds **complexity** & **costs** to your system!

# Design For Interoperability

**When does it apply? Why is it important?**

- When building systems of systems
- When using other services or providing services to others

# Design For Interoperability

- How to **generate** designs for interoperability?
  - **Create Shared Interfaces / Data Formats**
    -> Syntactic Interoperability
  - **Define the Semantics of Shared Data** to avoid **mars climate orbiter failure!**
    -> Semantic Interoperability

Carnegie Mellon University

# Design For Interoperability

- How to **communicate** designs for interoperability?
  - **Interface Descriptions** (e.g., OpenAPI)

| | |
|---|---|
| **Syntactic View** | Describe document format, the actions that can be performed, their parameters, and outputs. |
| **Semantic View** | Describe the purpose / meaning of the resource / action:<br>• **Side-effects**: Changes to the state of a resource or environment<br>• **Usage restrictions**: Who can perform this action?<br>• **Error Handling**: What errors can occur and why?<br>• **Examples**: Examples of outputs for a given input |

Carnegie Mellon University

# Design For Interoperability

How to **evaluate** designs for interoperability?

- Evaluation of an Implementation: Measure The **Percentage** of **Data** that has been **Exchanged Correctly**
- Evaluation of a Design: Measure the **Effort to Implement** the Interface in all Systems / Components
- Evaluation of a Design: Measure the **Variability** Allowed by the Interface / Format

# Design For Interoperability

How does Interoperability relate to … ?

- **Reusability**?
  - Interoperability allows you to “reuse” a **running** system rather than integrating the code into your system
- **Changeability**?
  - Fixed interfaces often **limit changeability**
- **Performance**?
  - Serialization and deserialization of data could add a **small runtime overhead** that is often not significant

# Design For Testability

How to **evaluate** designs for testability?

- **Controllability** measures how easy it is to **provide** a program or component with the **needed inputs**, in terms of values, operations, and behaviors, and bringing it into the desired **state** that should be tested.
- **Observability** measures how easy it is to **observe** the **behavior** of a program or component in terms of its outputs, quality attributes, effects on the environment, and other hardware and software components.

# Design For Testability

How to **generate** designs for testability?

- **Mock Components** verify indirect outputs via assertions
- **Test Stubs** control indirect inputs
- **Test Spies** verify indirect outputs via logging
- **Test-driven Development** ensures all written code is easily testable by writing tests before implementation
- **SOLID Principles** ensure code is easily testable

# Design For Testability

How to **communicate** designs for testability?

- Via test cases ☺

# Design For Testability

How does Testability relate to … ?

- **Changeability**?
  - They **both support each other**

# Design For Reuse?

**When does it apply? Why is it important?**

- Reuse **saves implementation effort**
- Reusable modules are **easier to understand**
- Reused modules tend to have **higher software quality** / **fewer defects**

Carnegie Mellon University

# Design For Reuse

How to **evaluate** designs for reusability?

- **Reuse Scenarios**
  - Unit of Reuse, Context of Reuse, Type of Adaptation, Effort of Adaptation

Carnegie Mellon University

# Design For Reuse

How to **generate** designs for reusability?

- **Simple, Well-Documented Interfaces:** **Reduce** the **complexity of the interface and the assumptions** the package makes about input data, actions, and environment
- **Loose Coupling**: Each module should **depend on as few** components as possible. Dependencies should be **explicit** and **minimize** assumptions.
- **High Cohesion**: Elements within a module should **work together** to fulfill a **single, well-defined purpose.**
- **SOLID Principles** ensure code is more reusable
- **Minimize AT-Modules, Maximize 0-Modules**
- **Avoid Dependencies from Large & Complex A Modules to T Modules**
- **Reduce Coupling to Frameworks**

# Design For Reuse

How to **communicate** designs for reusability?

- Description of Reuse Context
- Module Views
- Interface Descriptions

# Design For Reuse

How does reusability relate to … ?

- **Changeability**?
  - They **both support each other**
- **Testability**?
  - They **both support each other**
- **Performance**?
  - More reusable designs can, in some cases, be **slightly slower**

Carnegie Mellon University

# Design With Reuse

How to **generate** designs with reuse?

- **Identify Violated Assumptions of reused package** to avoid Ariane 5 rocket launch failure
- **Strive for Fewer Package Dependencies** to avoid the `left-pad` disaster
- **Keep Versions of Your Dependencies Fixed** to avoid **API-breaking changes**
- **Update Your Dependencies To Receive Bug Fixes**

Carnegie Mellon University

# Design With Reuse

How to **evaluate** a potential reuse candidate?

- **Cost-Benefit Analysis:**
  **Effort to adapt** the reusable module **vs.** **Effort saved** reusing the module

Carnegie Mellon University

# Design Process

How to **design in agile Projects?**

- Follow a **Risk-Driven Approach** to minimize unnecessary upfront design while still tackling high-priority risks
- Focus on **Changeability** to “respond to change” and to delay important decisions
- Maintain a **technical dept backlog** to keep track of design compromises

# Design Process

How to consider the **Human Aspect of Software Design?**

- Don’t Design In an Isolated Ivory Tower
- Design is a Collaborative, Hands-on Activity
- Combine Rational and Intuitive Decision Making

Carnegie Mellon University

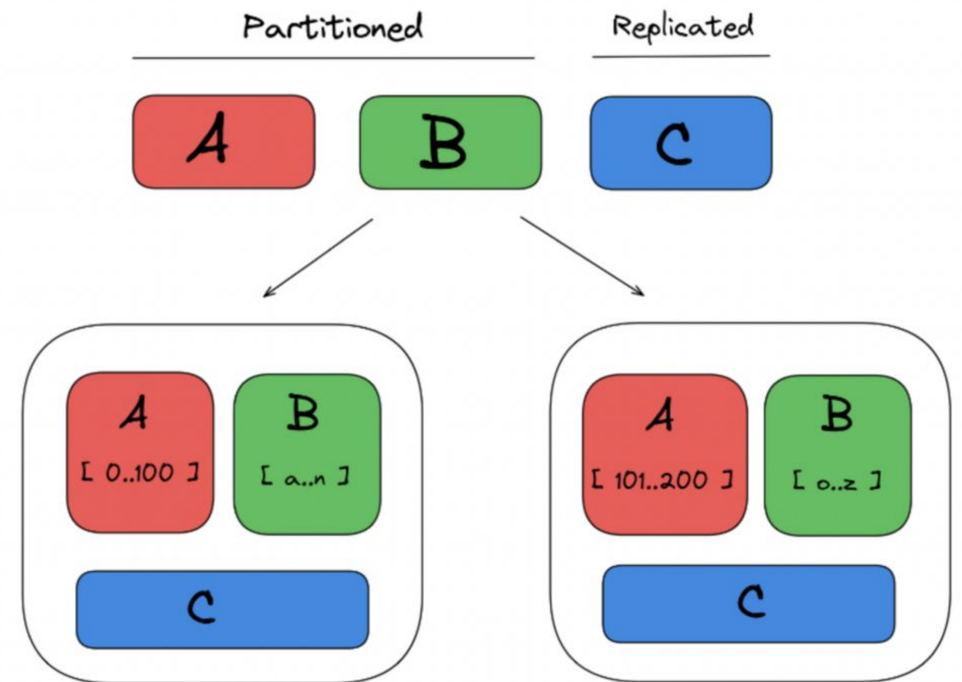
# Design Process

How to **Adjust the Design Process To Domain-Specific Needs?**

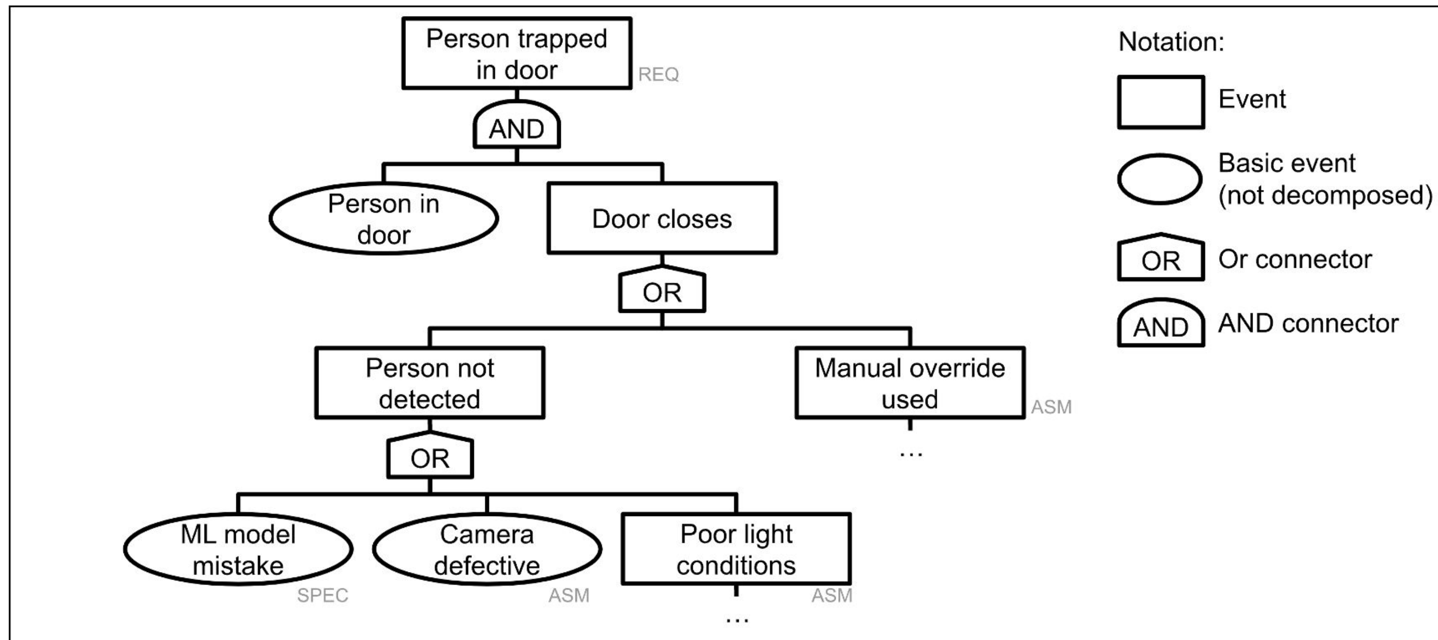
- **Higher risk domains** need more upfront design than lower risk domains
- **Longer projects** need more design documentation to keep track of previously made decisions

# Design for Scalability

- **Scalability**: Ability to handle growth in the amount of **workload** while maintaining an acceptable level of **performance**
- Design decisions: Vertical vs. horizontal scaling (increase capacity), load balancing (distribute work), caching (reduce bottlenecks)
- The "right" decisions for scalability depend highly on patterns of workload
- Delay investing in scalability until it's necessary!

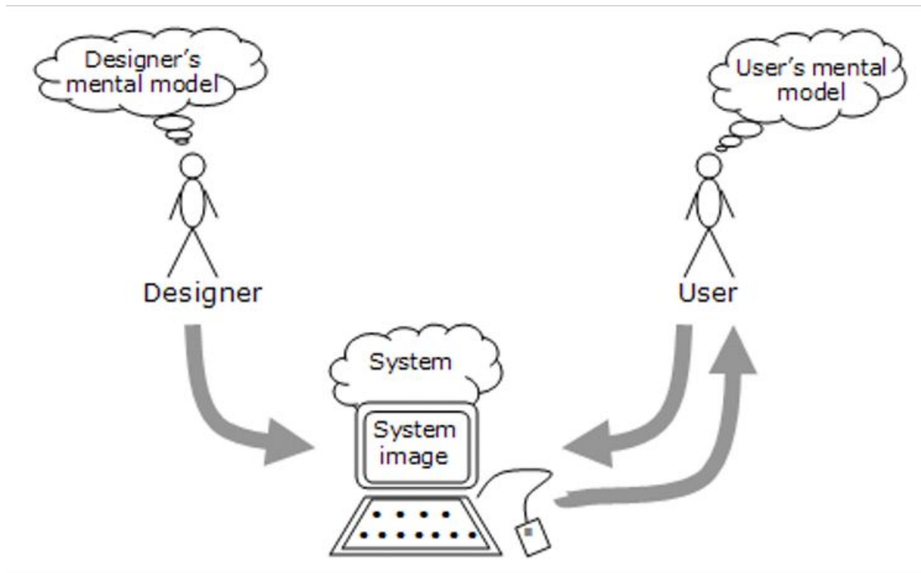


# Design for Robustness



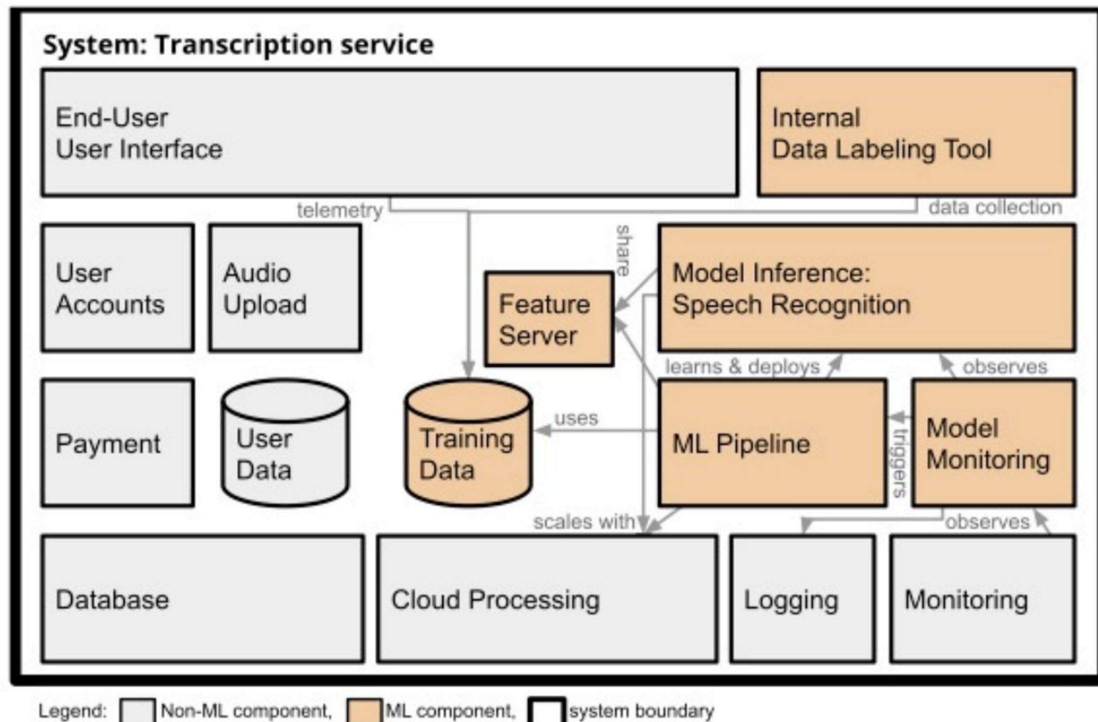
- **Robustness**: Ability to provide an **acceptable level of service** even when it operates under **abnormal conditions**
- No system will ever be “correct”: Be ready for things going wrong!
- Identify possible faults using fault tree analysis & HAZOP
- Apply robustness patterns: Guardrails, redundancy, degradation…

# Design for Usability



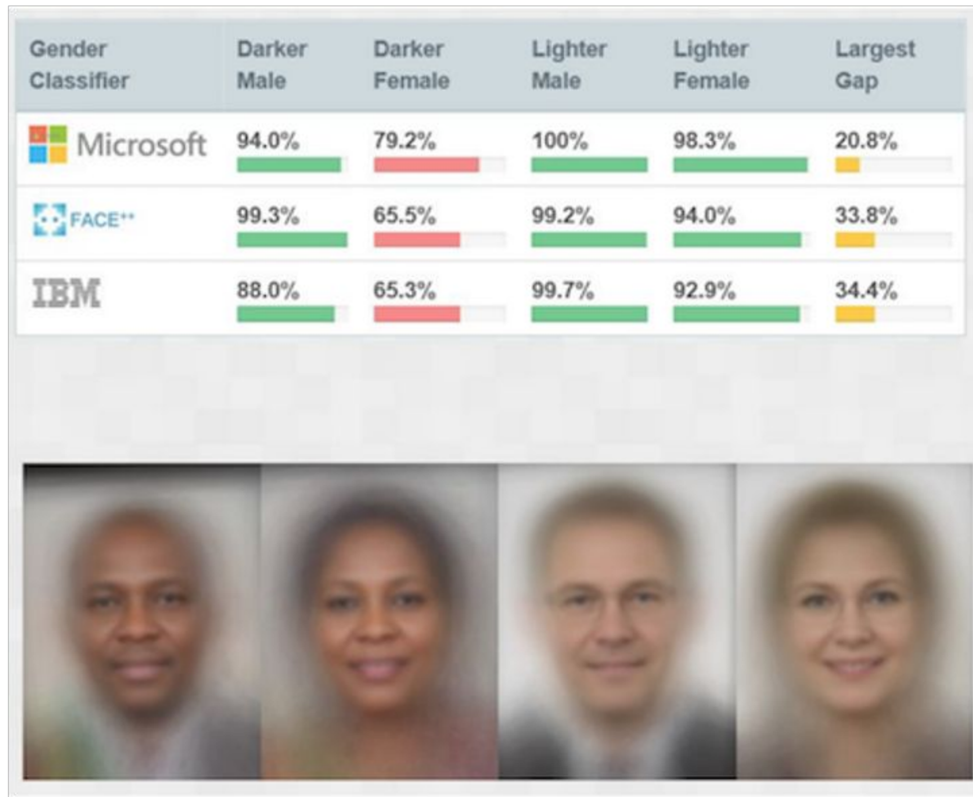
- **Mental model**: A person's understanding of how system works
- Mental model **mismatch** can cause confusion, increase user's effort and errors, lead to accidents…
- Identify user's mental model through similar apps or usability testing
- Design for **alignment**; help user build correct mental model through onboarding and explanations

# Design for AI-Enabled Systems




- Special design considerations are needed, especially for data curation, training, and model monitoring
- Accuracy is not the only important quality of an ML model!
- Ultimately, ML models are just one type of components within a larger software system; design principles & methods from other lectures still apply!

# Ethical and Responsible Design



| Gender Classifier | Darker Male | Darker Female | Lighter Male | Lighter Female | Largest Gap |
|---|---|---|---|---|---|
| Microsoft | 94.0% | 79.2% | 100% | 98.3% | 20.8% |
| FACE++ | 99.3% | 65.5% | 99.2% | 94.0% | 33.8% |
| IBM | 88.0% | 65.3% | 99.7% | 92.9% | 34.4% |

- Software engineers have power to influence users, our environment, and ultimately the society
- Identify **different groups of users** who may be affected
- Think of **possible harms** that can be caused by software
- Deliberately design the product to minimize harms
- **Consider**: Should I build this feature if potential harm is high?

# Future of Software Engineering?

## ChatGPT Will Replace Programmers Within 10 Years

Predicting The End of Manmade Software

Adam Hughes · Follow

Published in Level Up Coding · 12 min read · Feb 28

**Q. Your thoughts?**

# Closing Thoughts

- There will always be new technologies that push the level of abstraction higher (better LLMs, higher-level languages, etc.,)
- But design principles and methods from this class have existed for a long time and will continue to be relevant
- None of these methods, out-of-the-box, will guarantee that your product will be successful
- Human judgement is still needed to decide when it makes sense to apply a certain principle/method
- But being **deliberate about design, considering alternative options,** and **communicating them effectively** will help you become a successful software engineer

# Project Presentation

- In class this Wednesday
- Reflect on the design decisions, process, and teamwork
- **15 min** per team: We will be strict about the time!
- Focus on content, not layout or visuals
- See the project document for more instructions

# Final Exam

- **Time**: 8:30-11:30 am, Friday, May 3
- **Location**: SH (Scaife Hall) 238
- Open book (but no LLMs or contact with other humans)
- Every topic from the semester is within the scope
- Similar in style to the midterm & homework questions: Given a case study system, generate multiple design options, evaluate them with respect to quality attributes, consider trade-offs, and justify your final decision