17-423/723: Designing Large-scale Software Systems

Course Review

April 21, 2025

Learning Goals

- Describe and apply key foundational concepts in software design
- Apply principles, techniques, and tools to design software systems for various types of quality attributes
- Reflect on the role of design in the future of software developments

Project Presentation

- 15 min **max**. + 3 min Q&A
  - We will be strict about the time limit!
  - See the project presentation guidelines for more detail
- Every team member must be present

Final Exam

- **May 1 (Thursday), 5:30-8:30 pm in GHC 4215**
- Covers everything in the class
- Similar to the midterm in structure
  - Open book, but no electronics
- Sample exam from last year posted
  - **Note**: Different from this year's materials!
  - Will go through in this week's recitation

Course Roadmap

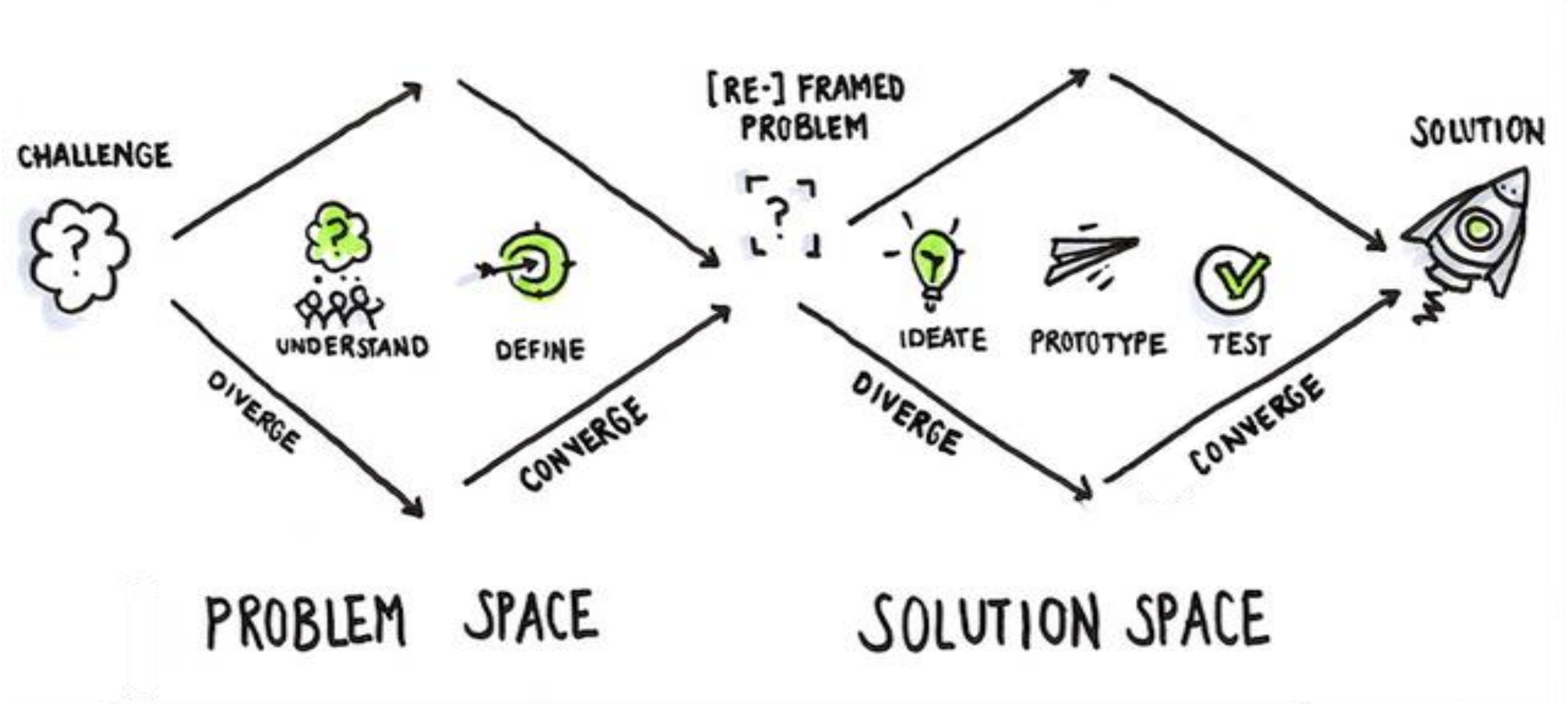
Foundational techniques and tools for design

Problem vs. solution space, domain & design modeling, quality attributes & trade-offs, interface specifications, design review

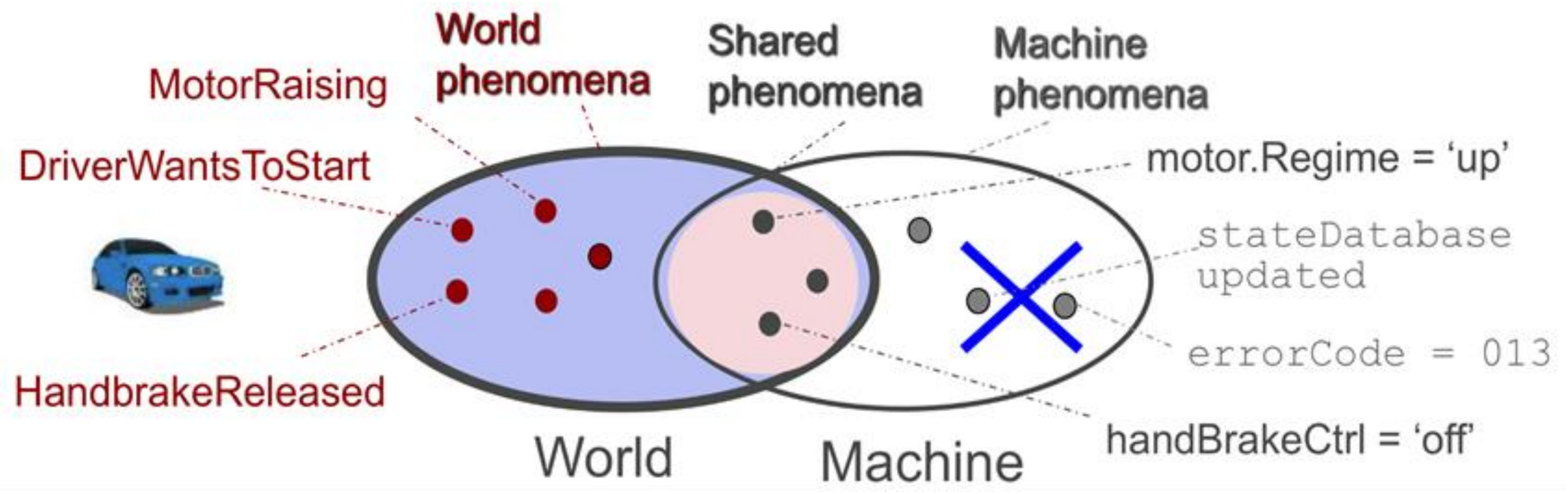
Designing for quality attributes

Design for change, testability, reuse, interoperability, scalability, robustness, security, AI, ethics

Problem vs. Solution Space



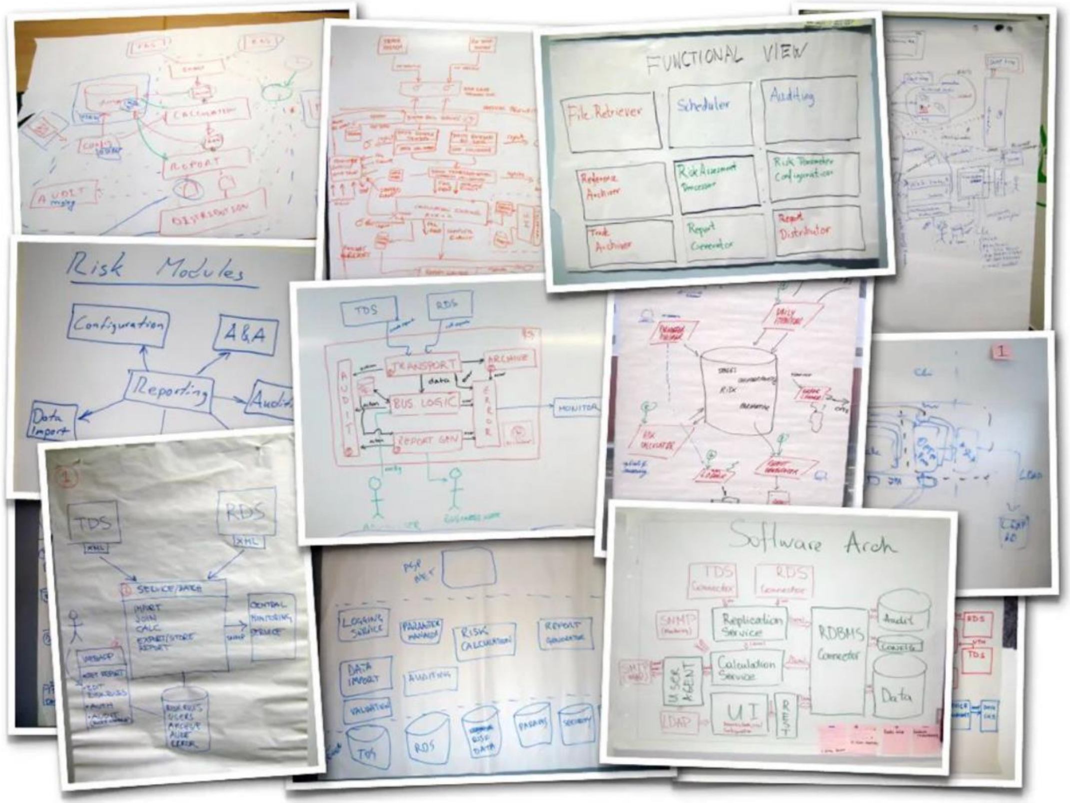
Problem vs. Solution Space



- 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 well-designed your software is)
- Identifying relevant parts of the world & assumptions is the 1st step to design

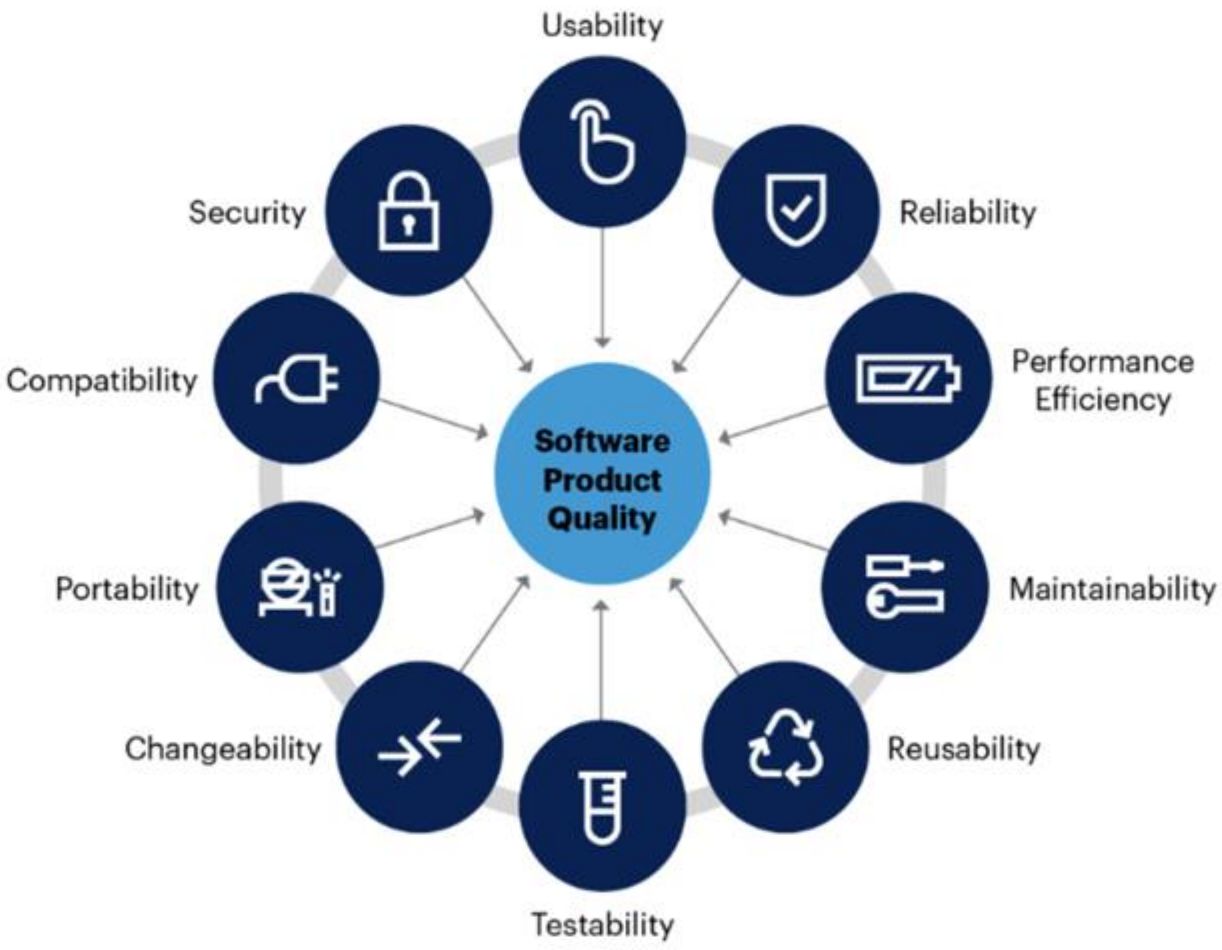
Design Abstractions

- Code is a poor way to convey design decisions
- Different abstractions (notations) are good at capturing different aspects of a design
  - Context models, component diagrams, data models, sequence diagrams
- Be precise & consistent with the meaning and use of a notation
- The goal is communication, not completeness; focus on design aspects that are most important

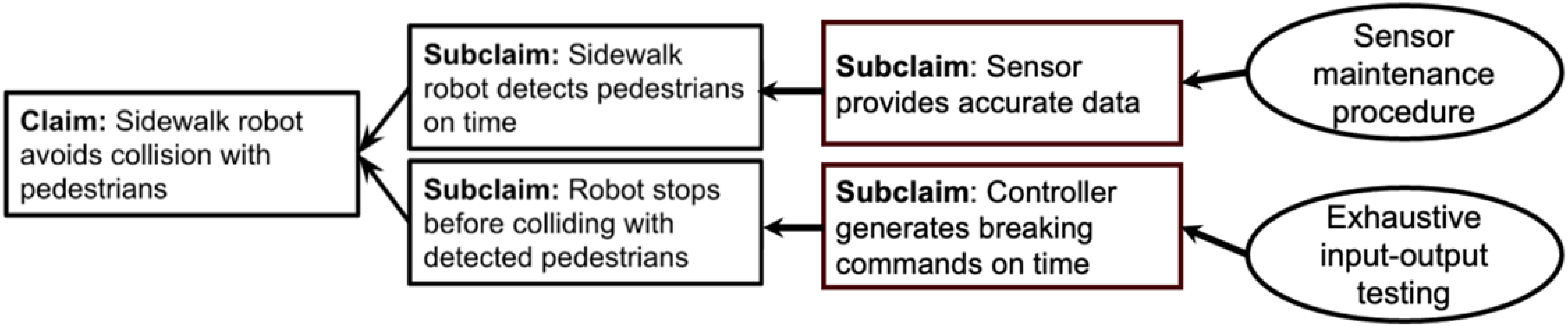


Quality Attributes (QAs)

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



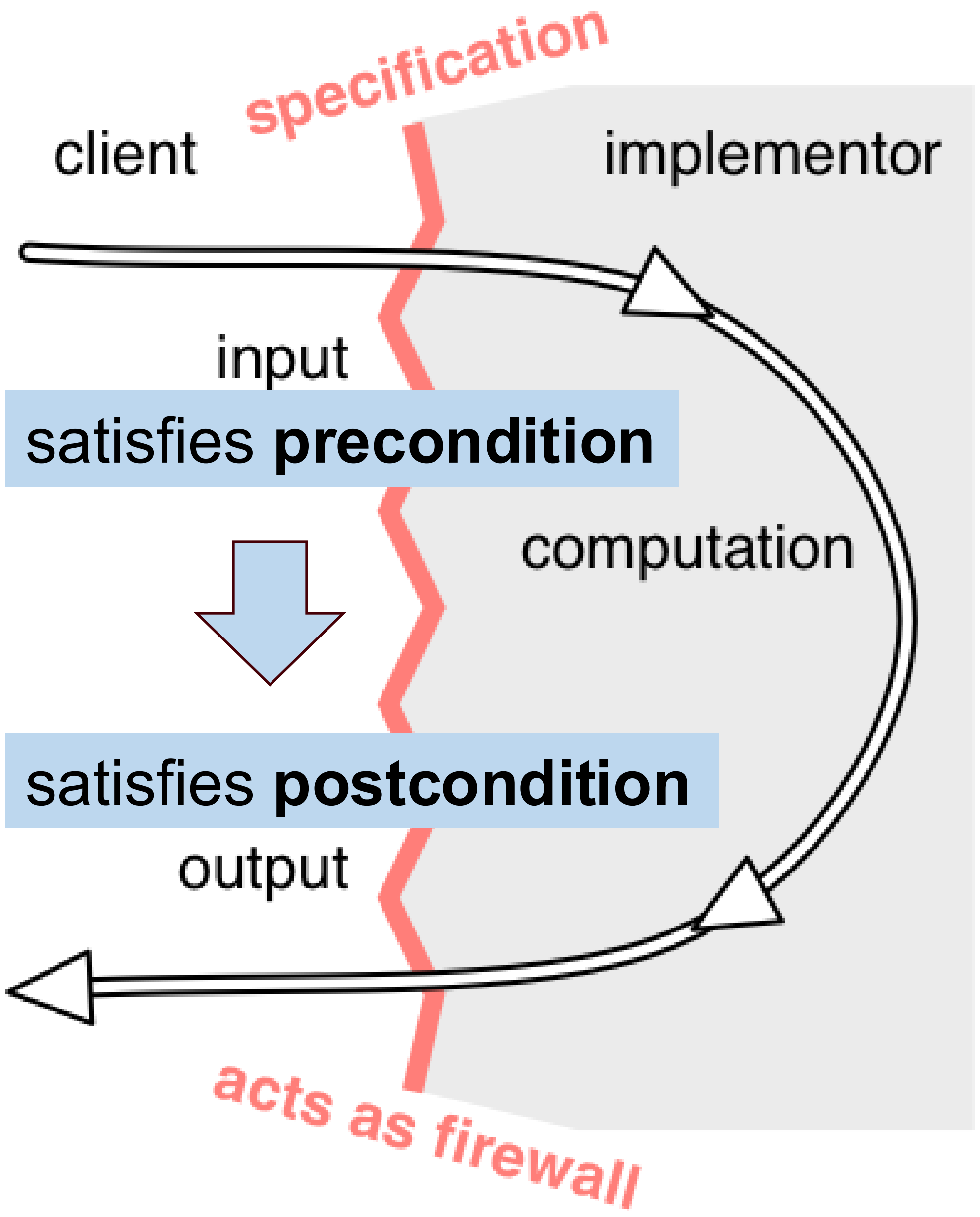
Arguing why your design works



- 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

Interface Specifications

- **Contract** between a client and a component
- **Pre-condition**
  - What the component **expects from the client**, expressed as a condition over the function input/component state
- **Post-condition**
  - What the component **promises to deliver**, as a condition over the function output/component state
- **Pre-condition ⇒ Post-condition** (i.e., logical implication)



Course Roadmap

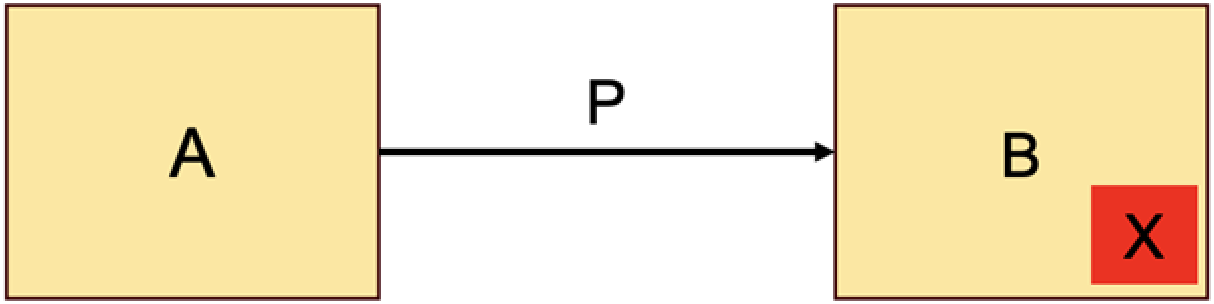
Foundational techniques and tools for design

Problem vs. solution space, domain & design modeling, quality attributes & trade-offs, interface specifications, design review

Designing for quality attributes

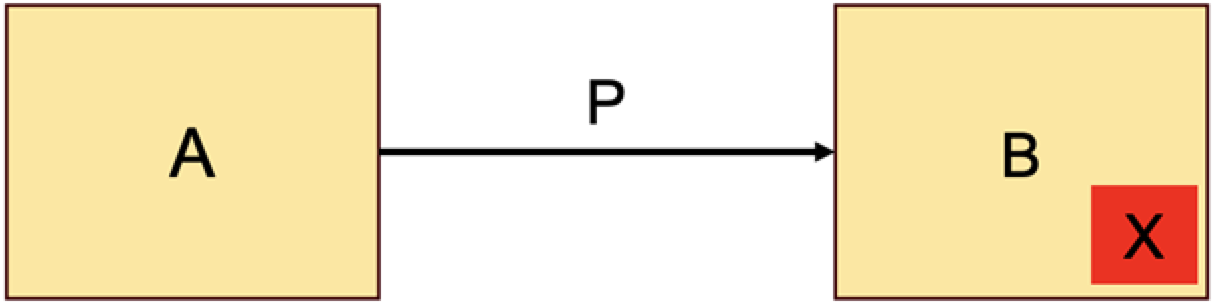
Design for change, testability, reuse, interoperability, scalability, robustness, security, AI, ethics

Design for Change



- **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 Change



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!

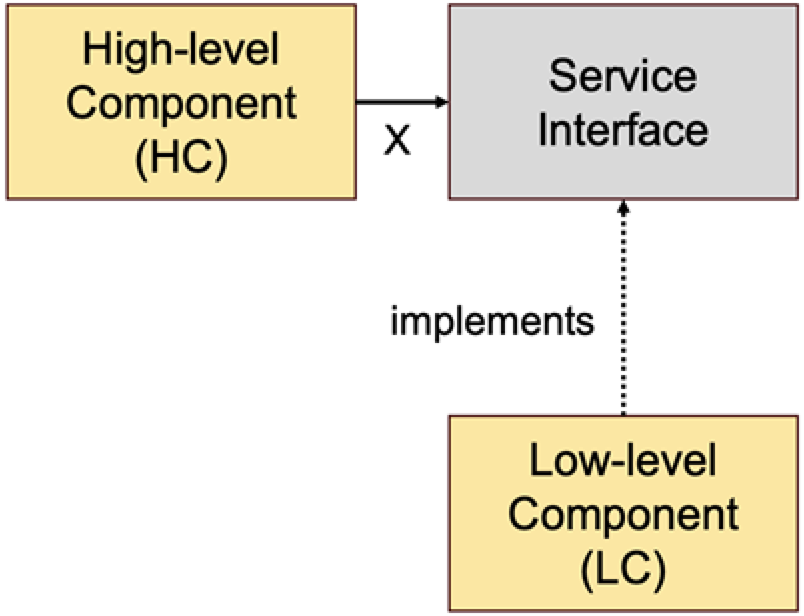
- **Changeability**: The amount of effort involved in making a particular change to a system
- **Key concept: Dependency** between components
  - Higher the degree of dependency, more you will need to change
- **Information hiding**: Hide secrets that are likely to change behind a component interface

Design for Change: SOLID Principles

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

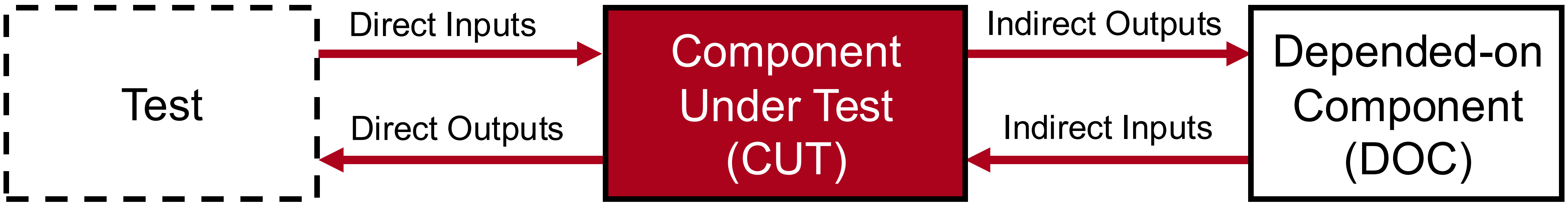
Design for Change: SOLID Principles

- **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



- **Don't over-modularize!** Consider (1) likely changes and (2) whether the flexibility to adapt to those changes is worth the cost

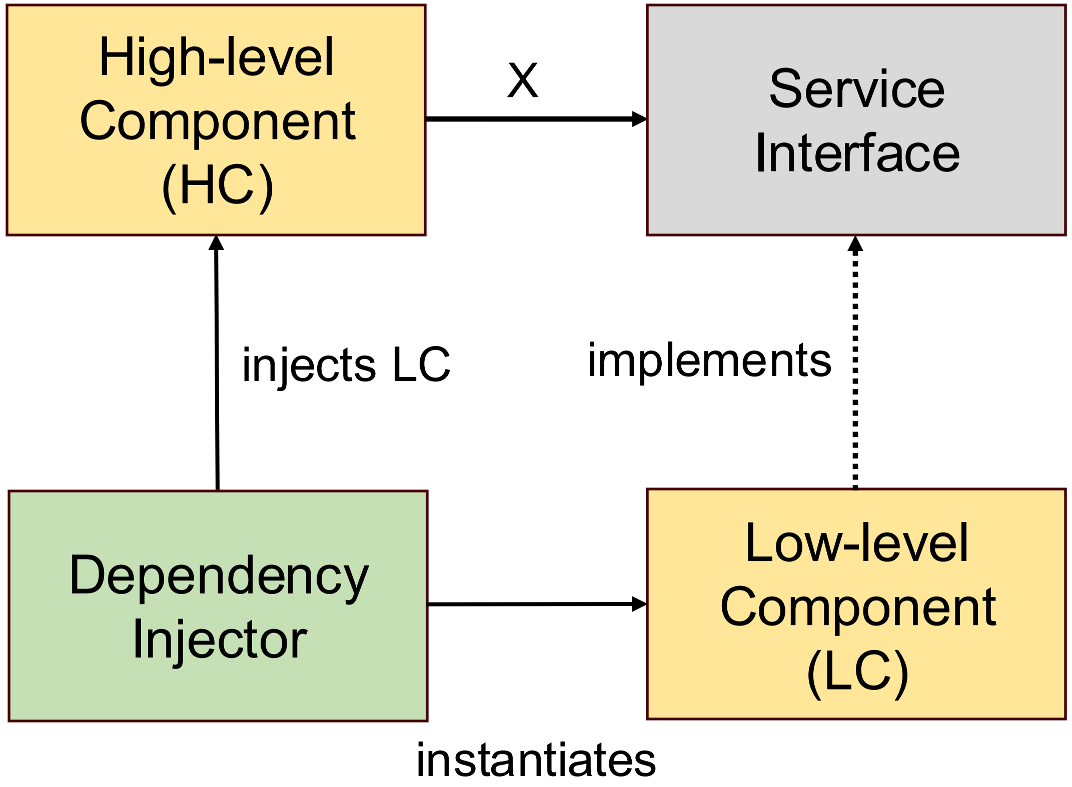
Design for Testability



- **Controllability**: How easy is it to bring a program to a particular state and/or inject it with a specific set of inputs?
- **Observability**: How easy is it to observe the behavior of a program, in terms of its outputs, quality attributes, or effects on its state?
- **Dependencies** can make testing difficult by reducing these two

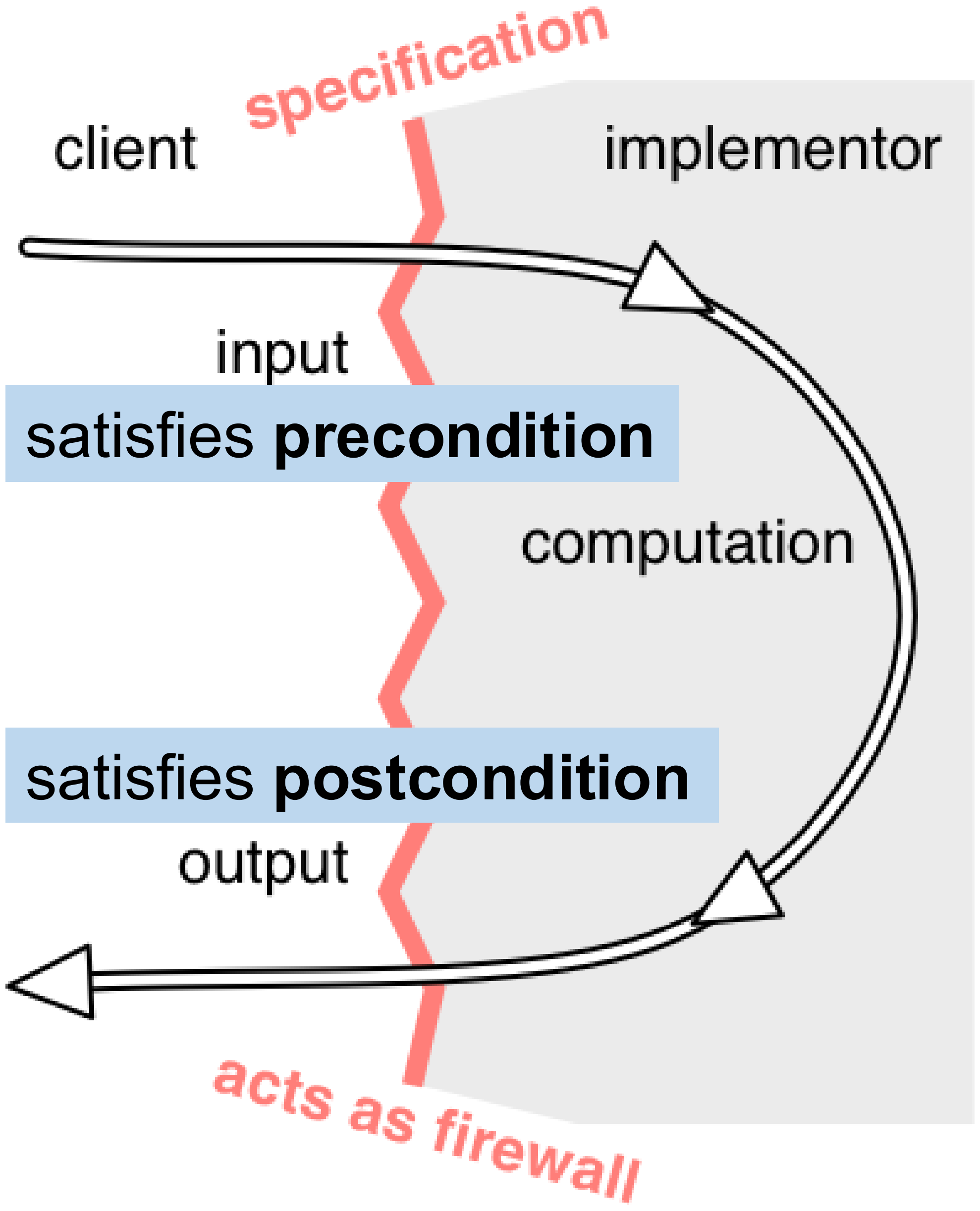
Design for Testability: Dependency Injection

- **Dependency injection:**
  - A component **receives** one or more components that it depends on
  - Dependencies are created and “injected” into the component by an **external** entity (i.e., client), instead of being created internally
- Improves **controllability** by separating the logic of creating dependencies
- **Q. Any potential downsides to dependency injection?**



Design by Contract (DbC)

- Check that a component and its client fulfill their contract by using **assertions**
  - At the *beginning* of a function, to check *pre-conditions*
  - At the *end* of a function, to check *post-conditions*



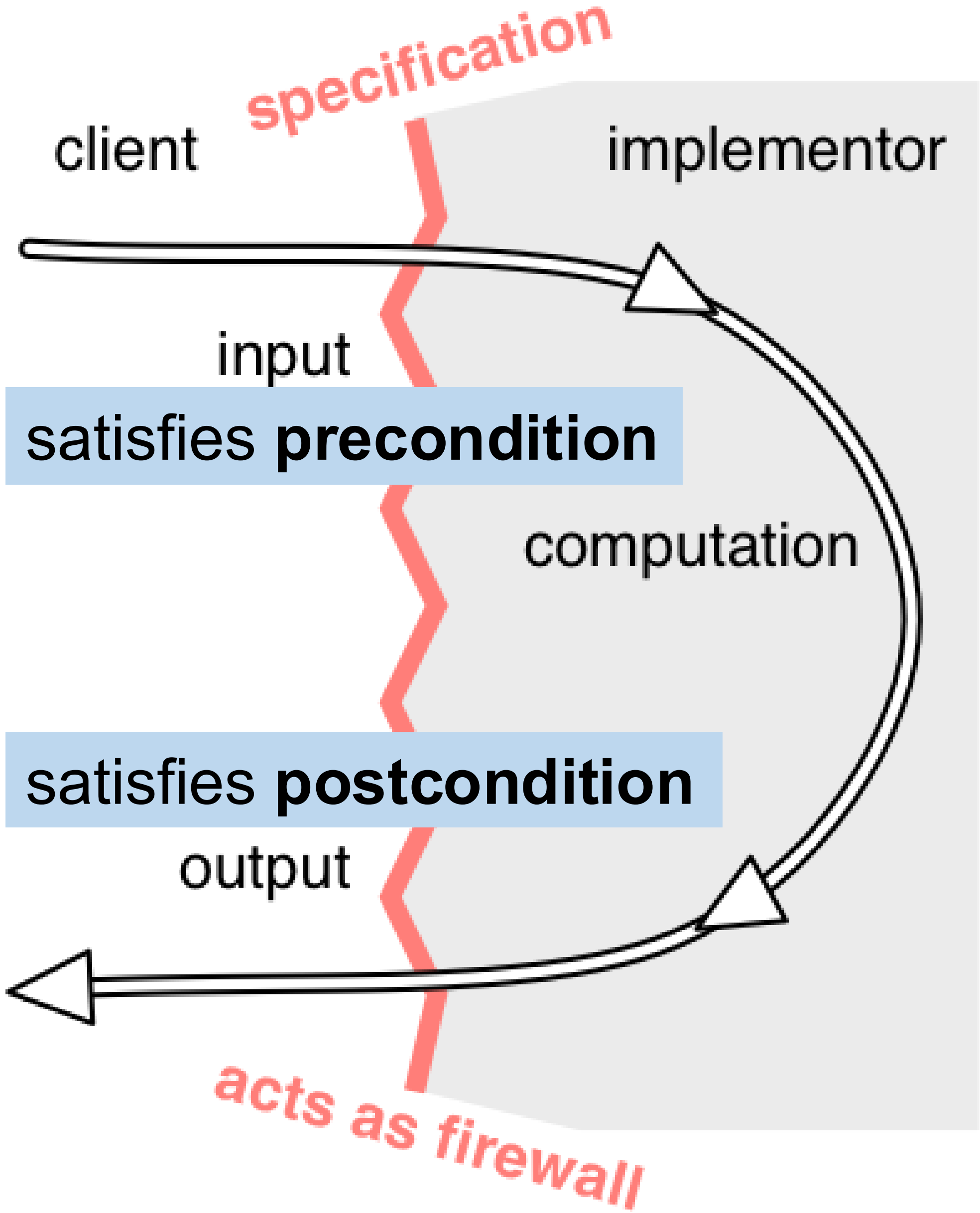
```java
public class Basket {
  private double totalValue = 0;
  private Map<Product, Integer> basket = new HashMap<>();

  // requires: product is not null; quantity is greater than 0
  // effects: product is added to the basket
  public void add(Product product, int qtyToAdd) {
    // check the post-condition holds on the exit
    assert product != null : “Product cannot be null”;
    assert qtyToAdd > 0 : “Cannot add 0 quantity”;

    // add the product
    // update the total value
    ...

    // check the post-condition holds on the exit
    assert basket.containsKey(product) :
           “Failed to add the product to the basket ;
  }
}
```

Assert that the pre-condition holds

Assert that the post-condition holds

Design by Contract (DbC)

- Check that a component and its client fulfill their contract by using **assertions**
  - At the *beginning* of a function, to check *pre-conditions*
  - At the *end* of a function, to check *post-conditions*
- **Invariant:** Condition that must hold throughout execution
  - Check in the initial state
  - Check after each function that modifies the system state



```java
public class Basket {
  private double totalValue;
  private Map<Product, Integer> basket = new HashMap<>();
  // invariant: totalValue is never negative

  // constructor
  public Basket() {
    // initialize the component state
    totalValue = 0;
    basket = new HashMap<>();
    // check that the component has been properly constructed
    // i.e., it satisfies the invariant
    totalValue >= 0;
  }
  // requires: product is not null; quantity is greater than 0
  // effects: product is added to the basket
  public void add(Product product, int qtyToAdd) {
    // add the product
    // update the total value
    ...
    // check that the method preserves the invariant
     totalValue >= 0;
  }
}
```

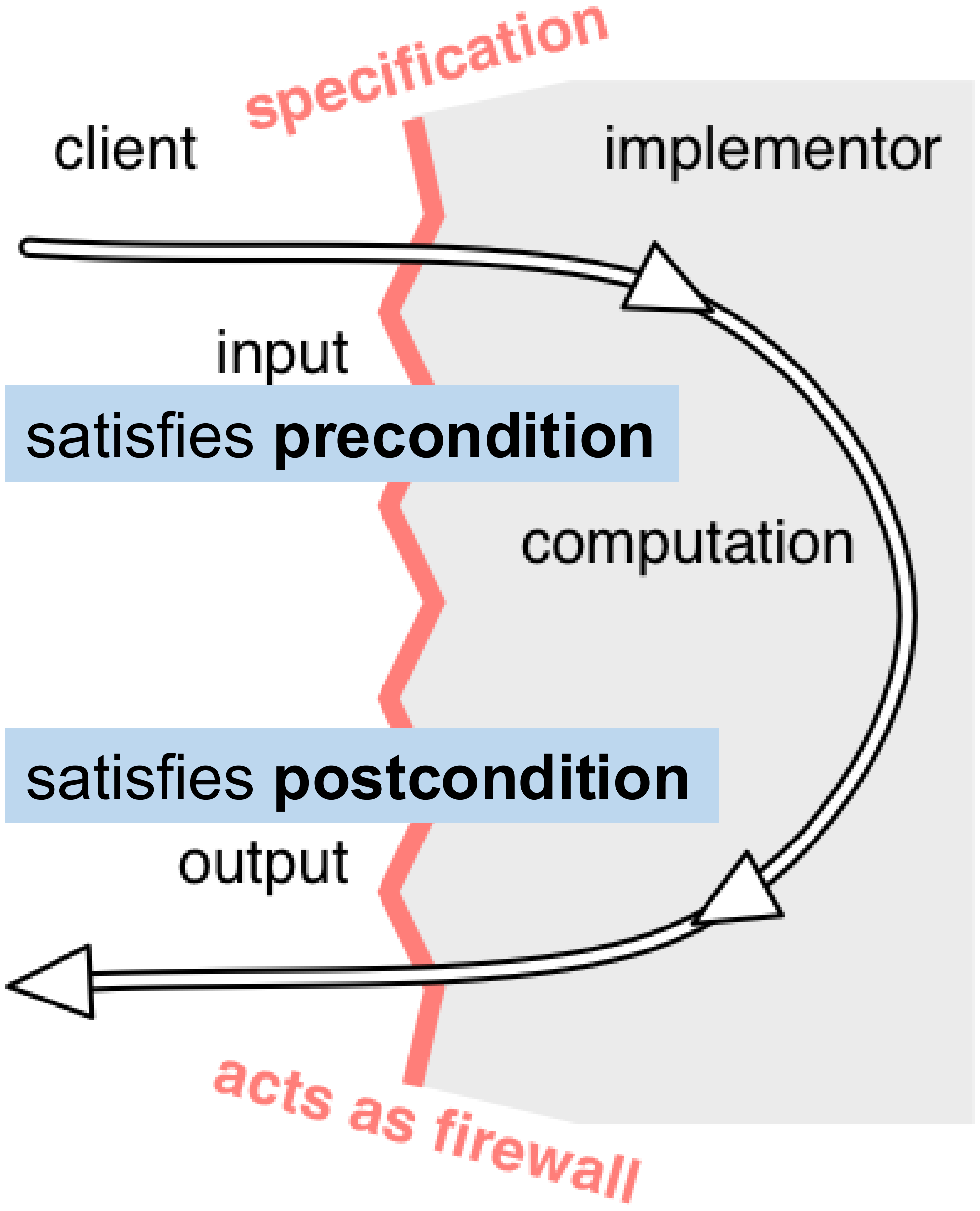
Invariant documentation

Check that invariant holds in initial state

Check that invariant holds in the post-state

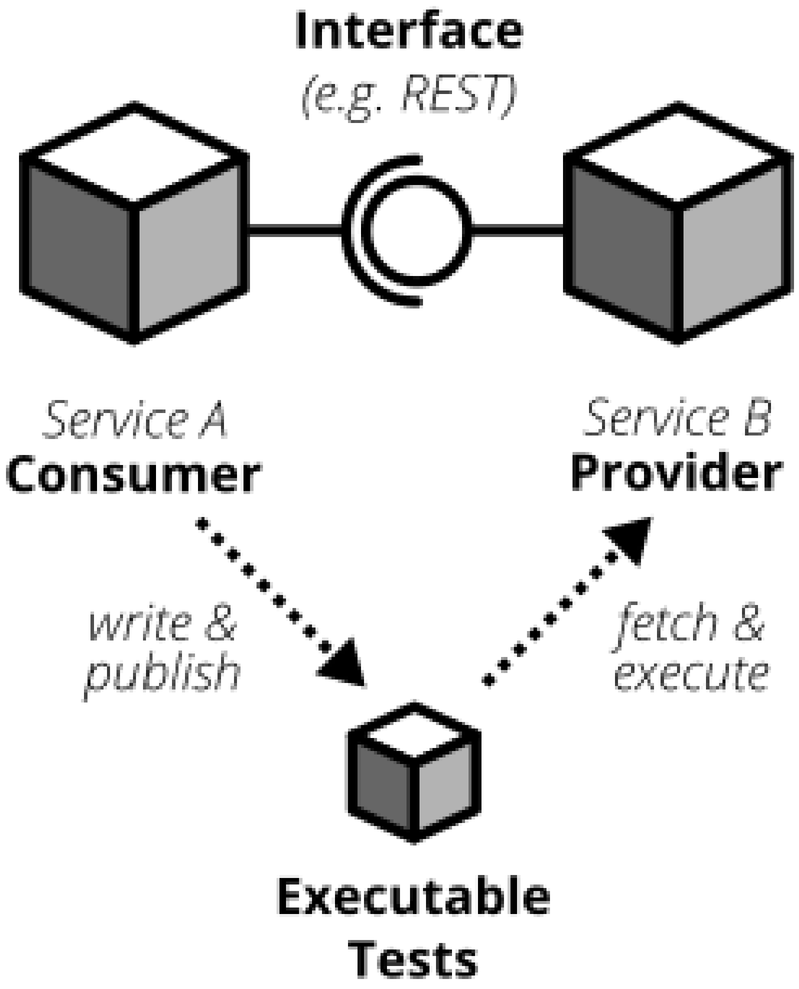
Design by Contract (DbC)

- Check that a component and its client fulfill their contract by using **assertions**
  - At the *beginning* of a function, to check *pre-conditions*
  - At the *end* of a function, to check *post-conditions*
- **Invariant:** Condition that must hold throughout execution
  - Check in the initial state
  - Check after each function that modifies the system state
- **Q. How is this different from testing?**



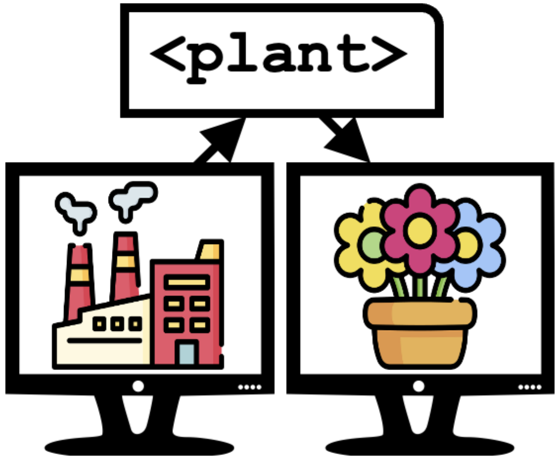
Contract Testing

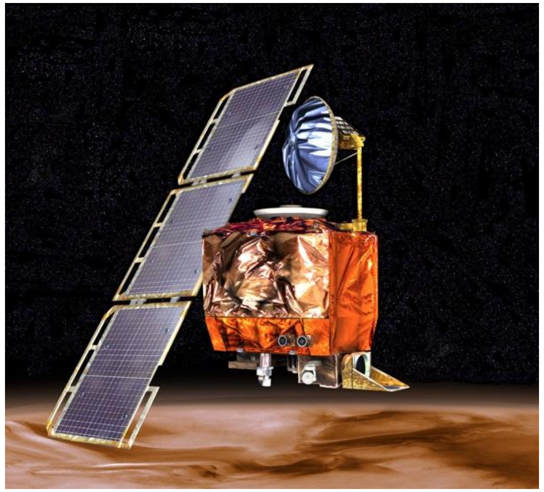
- An incremental, service-by-service approach to integration testing
- **Provider**: Provides data to consumers
- **Consumer:** Processes data obtained from a provider
- **Consumer-driven Contract (CDC):** Describes what the consumer expects from the provider as an output
- Allows services to be tested without having to run all of them
- When a provider changes, contracts can be used as regression tests, to detect whether the change affects its consumers



Design for Interoperability

- **Syntactic** interoperability: Multiple systems exchange data over a **shared format** & a **protocol**
- **Semantic** interoperability: Multiple systems exchange and assign a **common interpretation** to data
- An **ontology** defines concepts, their relationships, and constraints in an application area of interest
- Design to support **backward compatibility**

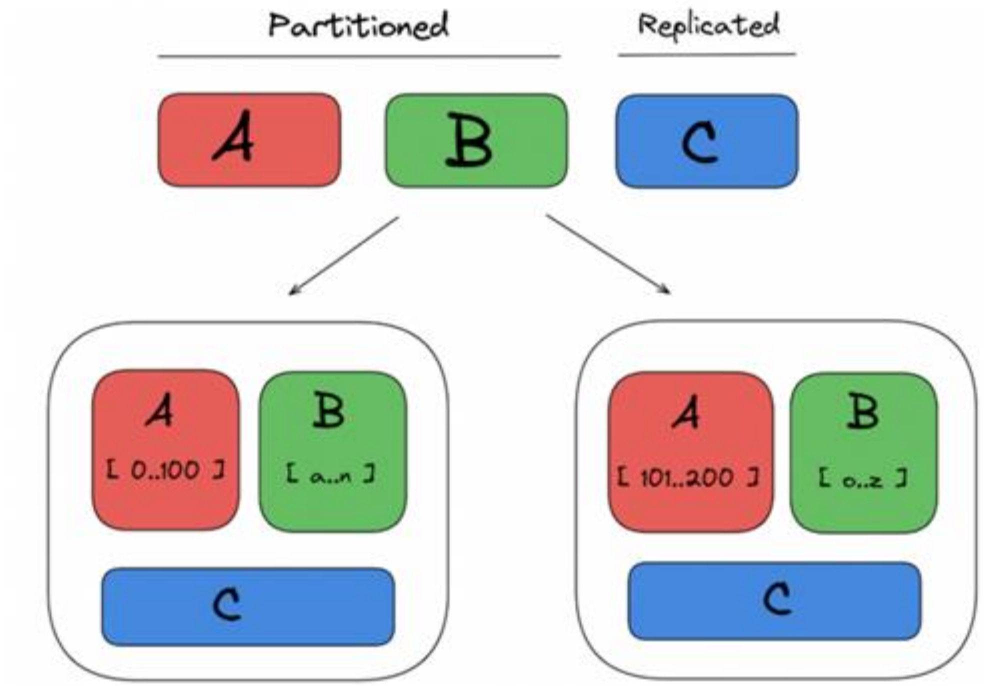




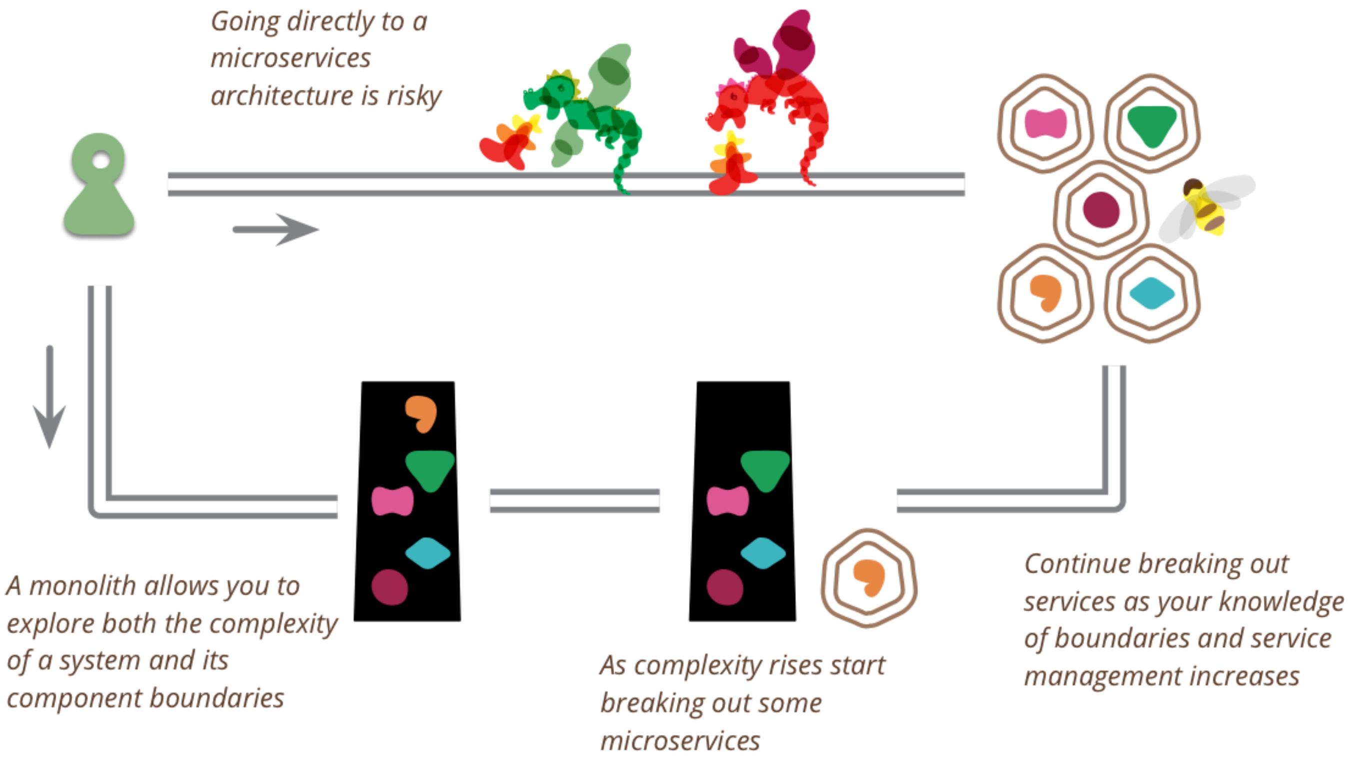
Mars Climate Orbiter

Design for Scalability

- **Scalability**: Ability to handle growth in the amount of **workload** while maintaining an acceptable level of **performance**
- **Design decisions**: Data model (to store data), vertical vs. horizontal scaling (increase overall capacity), load balancing (distribute work across machines), caching (reduce bottlenecks)
- In general, correctly designing a distributed system is really hard (recall: CAP theorem)



Design for Scalability

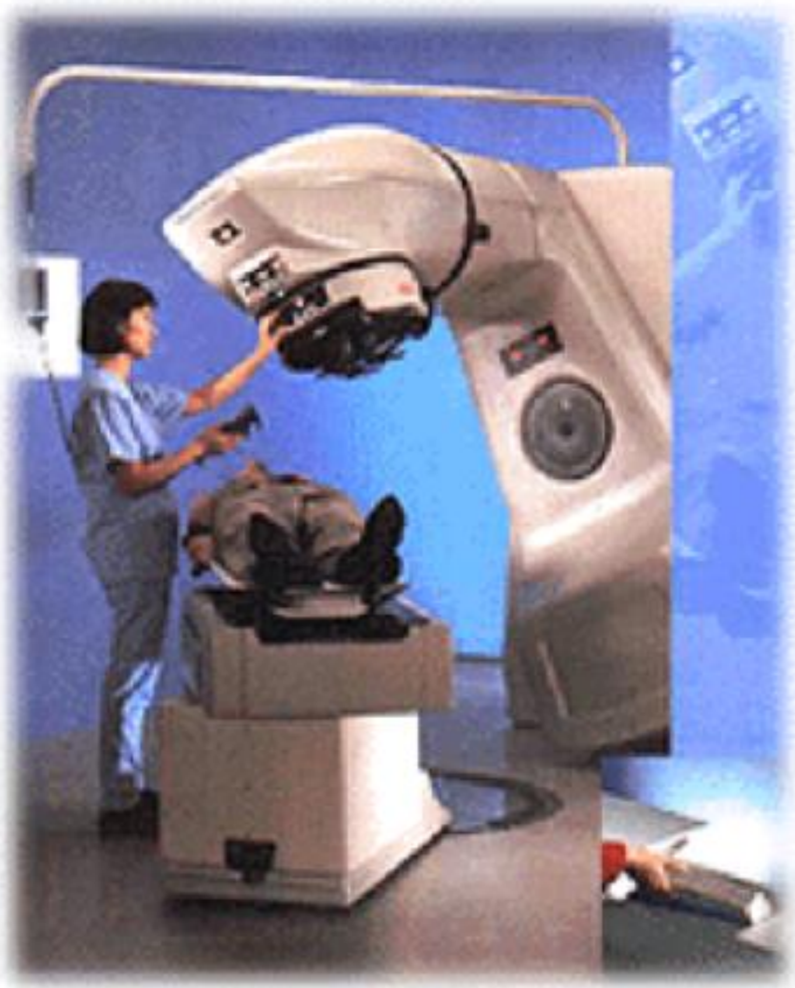


https://martinfowler.com/bliki/MonolithFirst.html

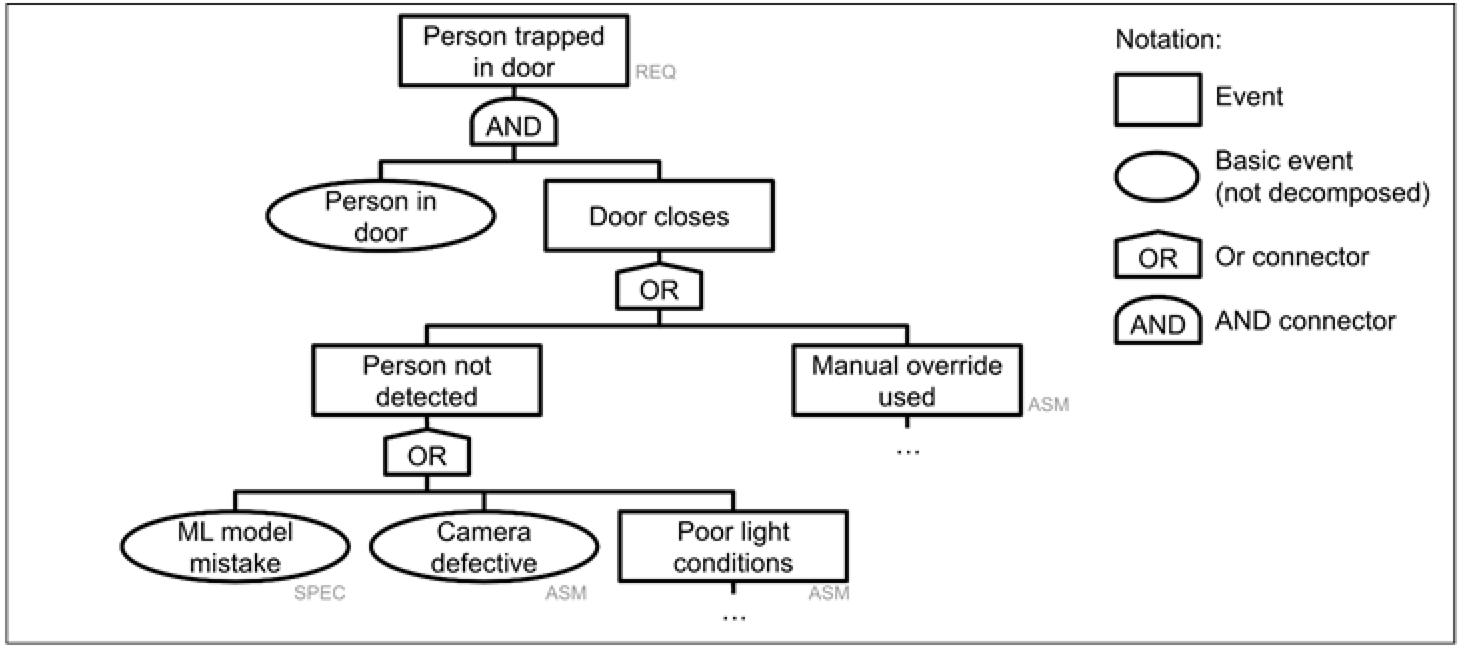
- "Right" decisions for scalability depend highly on patterns of workload
    - And you won't find these out until after you've deployed your system
- Delay investing in scalability until it's necessary!

Design for Robustness

- **Robustness**: Ability to provide an **acceptable level of service** even when it operates under **abnormal conditions**
- Many past accidents in software are due to lack of robustness against human errors or unexpected faults in the environment
- No system will ever be “correct”: Be ready for things going wrong!



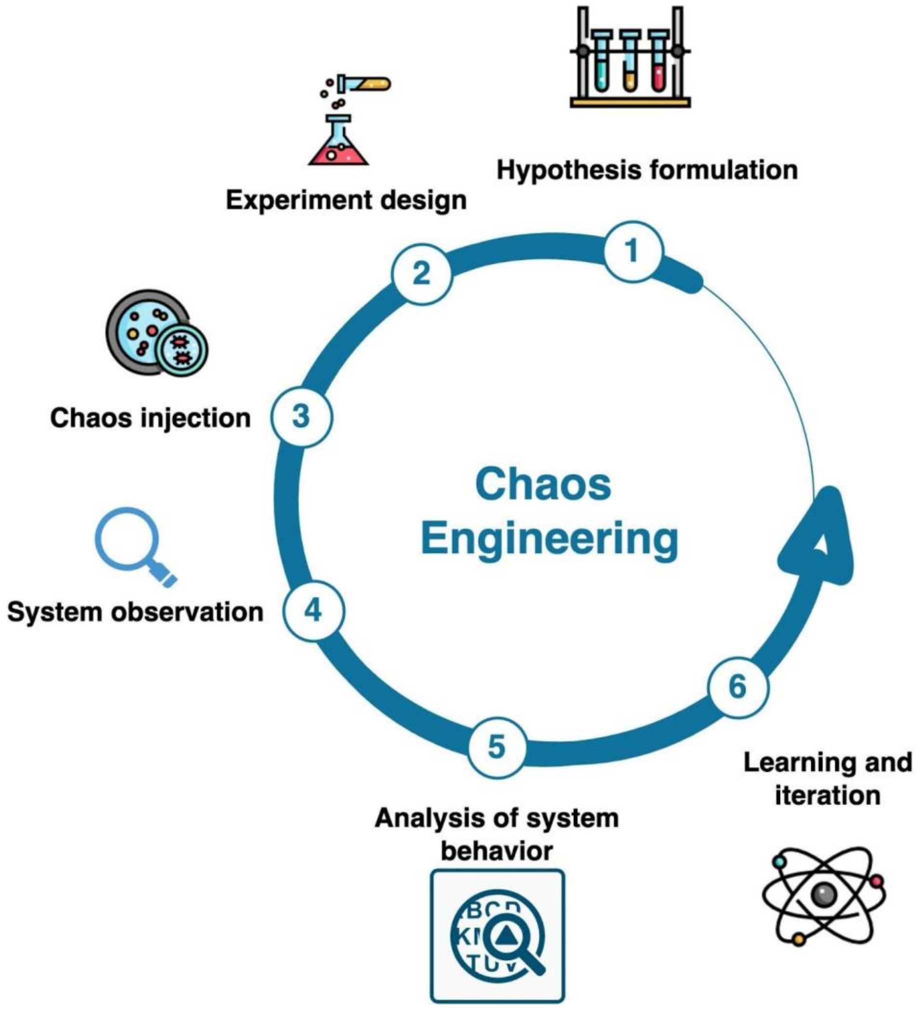
Design for Robustness



- Identify possible faults using **fault tree analysis** & **HAZOP**
- Apply **robustness patterns**: Guardrails, redundancy, separation, graceful degradation, human in the loop, undoable action

Chaos Testing

- Evaluate robustness **realistic** failures
  - Create a **hypothesis** about system behavior under a failure
  - Designate parts of the system as **control** vs. **experimental** groups
  - Inject a failure into the experimental group
  - Measure and compare a desired metric across groups
  - Improve the design to deal with the failure
- Encourages developers to deliberately design the system to be ready for failures!

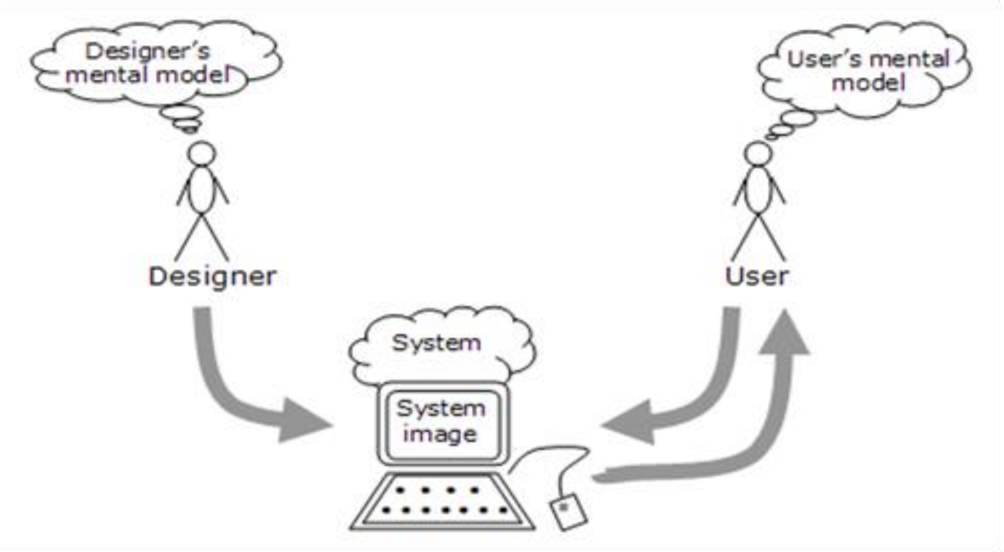


Design for Security

Threat	Desired property	Threat Definition
Spoofing	Authenticity	Pretending to be something or someone other than yourself
Tampering	Integrity	Modifying something on disk, network, memory, or elsewhere
Repudiation	Non-repudiability	Claiming that you didn't do something or were not responsible; can be honest or false
Information disclosure	Confidentiality	Someone obtaining information they are not authorized to access
Denial of service	Availability	Exhausting resources needed to provide service
Elevation of privilege	Authorization	Allowing someone to do something they are not authorized to do

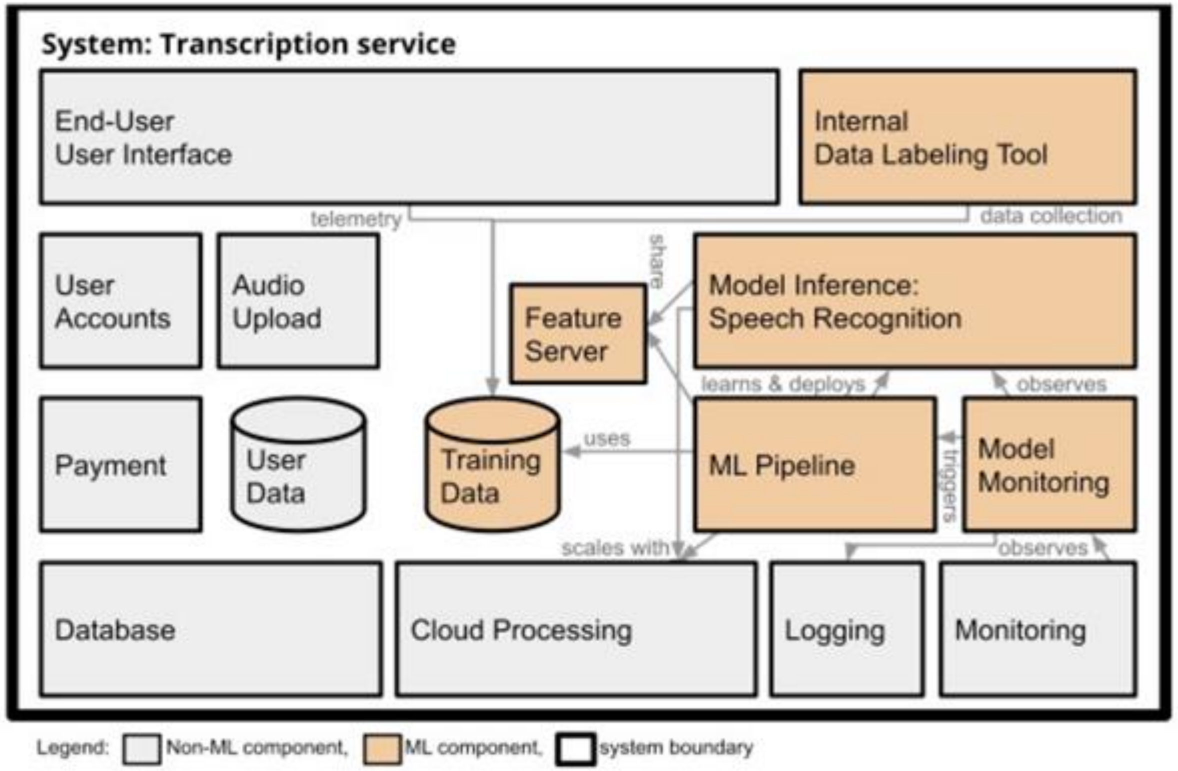
- **C**onfidentiality, **I**ntegrity, **A**vailability ("CIA") requirements
- **Threat modeling**: Estimate an attacker's possible actions
  - **STRIDE**: A systematic approach to identify possible threats
- **Principles:** Least privileges, open design, reduce trusted computing base (TCB), don't invent your own security methods

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…
- Design for **alignment**: Don't invent a new UI unless necessary
    - Identify user's mental model through similar apps or usability testing
    - Help users adapt their model through onboarding & transparency

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, 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?

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, 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?

Reflections on Design

Is Design Worth Doing?

- **Q1. What are some benefits of doing an explicit design before writing code?**
- **Q2. What makes designing software particularly challenging?**
- **Q3. Designing a perfect system is difficult/impossible, so what's the point anyway?**

Viewpoint: Design as Uncertainty Reduction

- Many forms of uncertainty in software
  - Will users like our product? Will it be usable? Will it run fast enough? Will it be secure against attacks x, y, z? …
- Explicit design enables early prototyping & evaluation
  - Can identify & rule out bad designs early on
  - (This can be done in both waterfall & agile methods!)
- Explicit design enables better management of **technical debt**
  - Cost of additional rework caused by choosing an early (limited) solution
  - E.g., “We will just add encryption later to make the system secure”
- The goal is not to design the perfect product upfront, but to reduce the amount of cost possibly incurred later in the product cycle

“The designers usually find themselves floundering in a sea of possibilities, unclear about how one choice will limit their freedom to make other choices...

There probably isn’t a ‘best’ way to build it, or even any major part of it; what’s important is to avoid a terrible way, and to have clear division of responsibilities among the parts.”

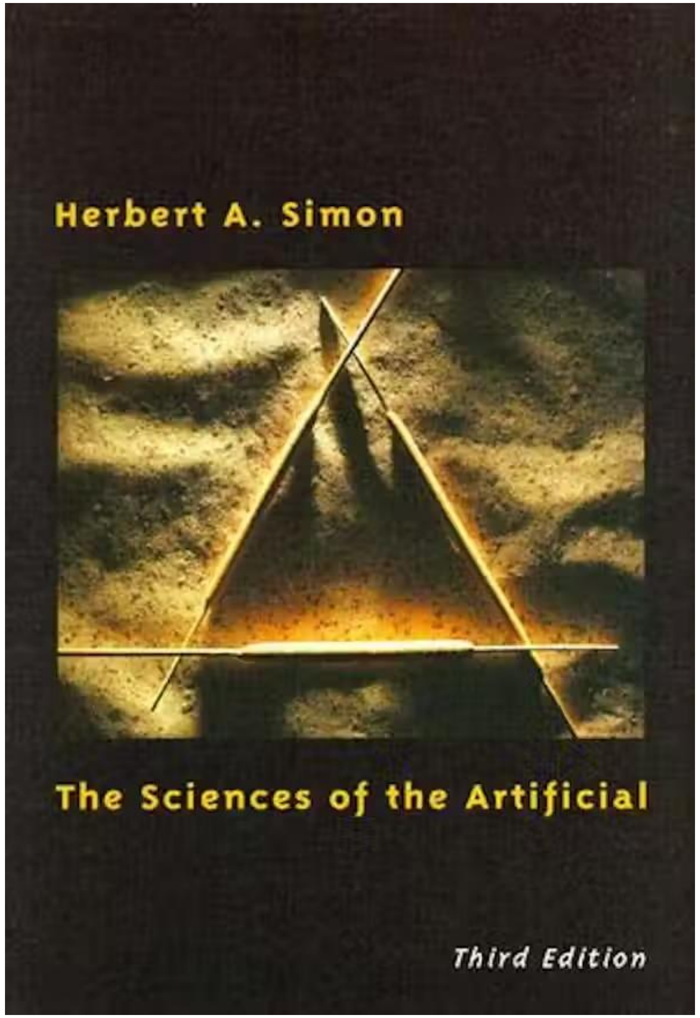
Butler Lampson
Hints for Computer Systems Design

Viewpoint #2: Design as Resiliency Building

- **Resiliency**: Ability of a person or a system to tolerate external disturbances and bounce back
- **Resiliency in system**
  - Changeability, scalability, robustness, security – these are about dealing with different types of disturbances to the system
  - If you don't design resiliency into the system, unlikely that this property will "emerge" by itself
- **Resiliency in designer**
  - By studying & evaluating alternative designs, the designer will develop an in-depth understanding of the problem & solution spaces
  - This knowledge is crucial for maintaining & evolving the system as requirements inevitably change over time
  - (Product trouble often begins when this person leaves the project!)

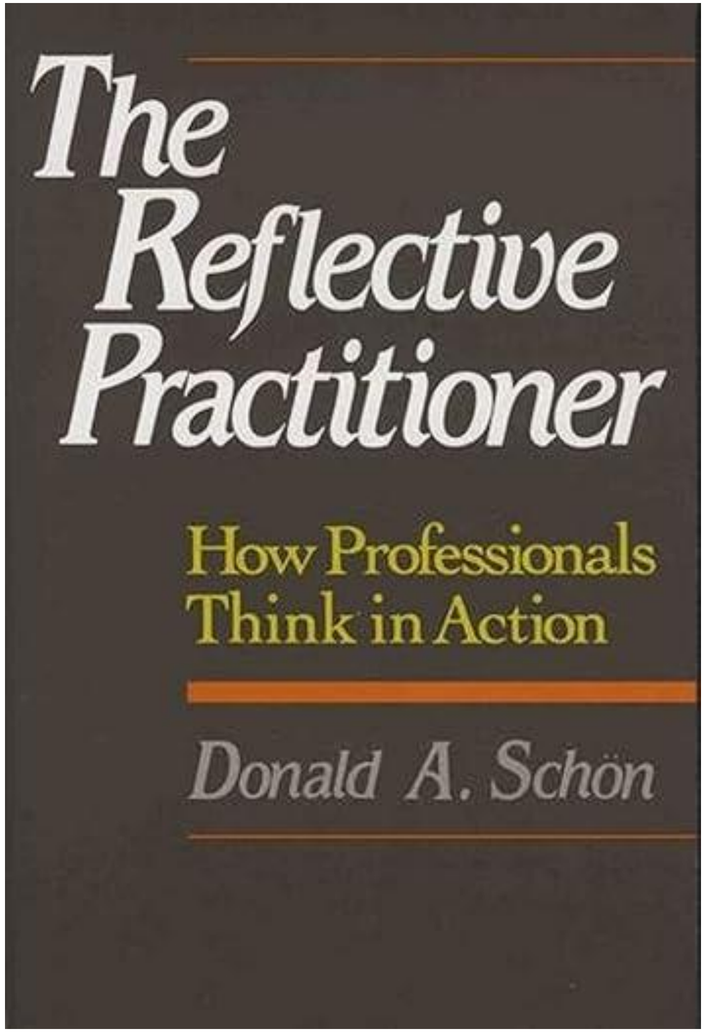
Perspective: Design as Problem Solving

- Design is a **systematic, rational process**
  - A description of a problem space & constraints (i.e., assumptions) is given
  - Designer makes a sequence of design decisions
  - Each candidate solution is evaluated until a satisfactory design is found
- Simon hinted that one day, this process could be automated by computers



Perspective: Design as Problem Setting

- Design is a **conversation between the problem space & the designer**
  - Simon's model is flawed; designers don't actually work like this in practice
  - As the designer explores possible solutions, they learn more about the problem itself
  - Outcome of design is both the product & also an increased understanding of the problem space
- This is unlikely to be fully automatable by computers



Simon vs. Scholn

- **Q. Which one do you think is the “right” model of the design?**

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

- **Q1. Will AI/LLMs replace software engineers in the future?**
- **Q2. Will software become more reliable and correct because of AI?**

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

Summary

- Exit ticket!