17-723: Designing Large-scale Software Systems

Design For Reuse

Tobias Dürschmid

Carnegie Mellon University

Why should I care about Reusability?

Reusability is Strongly Linked with **Understandability**

- **Program comprehension** takes up **58%** of professional developers' time [1]
- Goal: Designing modules that can be **used without understanding how they work internally**
- Design principles in this lecture do not only help to design reusable software, but also **understandable software!**

[1] Xia, Xin, et al. "Measuring program comprehension: A large-scale field study with professionals." *IEEE TSE* (2017)

Carnegie Mellon University

This Lecture - Reusability

- How to **Design Modules To Be Reusable**?
- How to **Design Complex Domain Logic To Be Reusable**?
- How to **Evaluate Reusability**?
- How does Reusability **Relate to Other Quality Attributes**?

Carnegie Mellon University

Reusability Requirements are Specified via **Reuse Scenarios**

Scenario

1. **Unit of Reuse** (modules)
2. **Context of Reuse** (who, where, when, how?)

Remember **Cost-Benefit Analysis** from the Lecture on Design With Reuse

Measure

Effort to Adapt to new Context

Type of adaption (e., configuration, code change, …)

Carnegie Mellon University

Example Reuse Scenario

Example Domain: Imagine Stylization Apps

Unit of Reuse

The **noise reduction image filter** of the pencil hatching app

Context of Reuse

should be reusable for **all other image stylization effects**

Effort of Adaption **Type of Adaption**

without making any changes to the source code.

Carnegie Mellon University

Example Reuse Scenario

Example Domain: Imagine Stylization Apps

Unit of Reuse

The **noise reduction image filter** of the pencil hatching app

Context of Reuse

Adds Performance Constraints

should be reusable for **processing of very large images**

Effort of Adaption

Type of Adaption

via **end-user-adjustable** **parameter configuration**.

Carnegie Mellon University

How to
Design Modules To Be Reusable?

Designing Large-scale Software Systems - Design For Reuse

Carnegie Mellon University

Example: **Unix Pipes** allow forwarding the output of one program into the input channel of another program

Pattern for Reuse: **Pipes & Filters**

Problem: How to build a system that **process data streams** in a **reusable**, **composable**, **flexible**, and **independently developable** way?

Solution: Divide a larger processing task into a sequence of smaller, independent processing steps (***Filters***) that are connected by channels (***Pipes***).

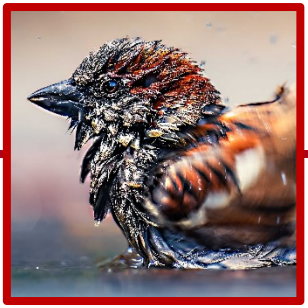
In-Class Activity: **Describe Reasons why Filters are Highly Reusable!**

Pipes have **Simple Interfaces**
(e.g., 2D Pixel Graphics)

Pattern for Reuse: **Pipes & Filters**

Each Filter **Does Only One Thing**
(e.g., increase brightness, reduce noise)

Noise Reduction Filter



Brightness Filter

Edge Detection Filter

Color Adjustment Filter

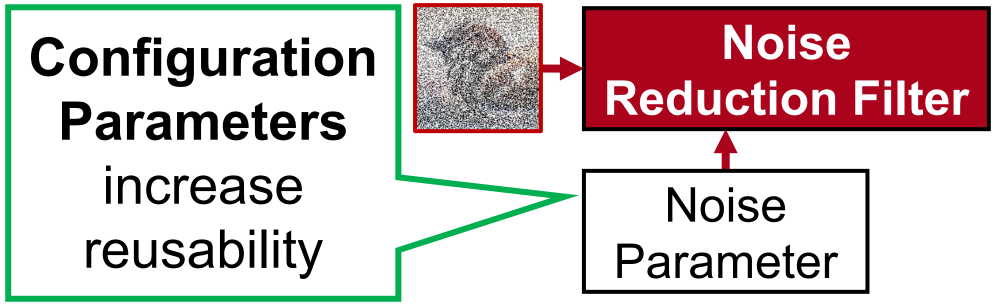
Filters are **Loosely Coupled**
(i.e., each filter can be connected to any filter)

Carnegie Mellon University

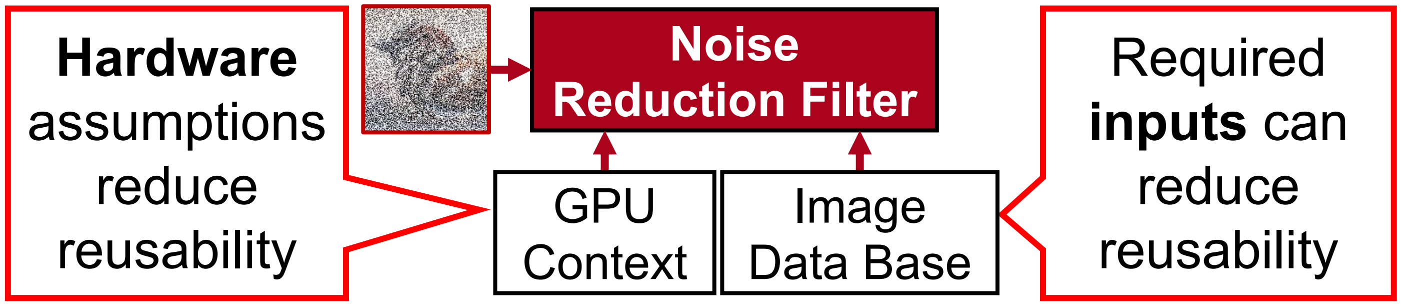
Design Principle for Design for Reuse: **Simple, Well-Documented Interfaces**

Assumes only a 2D pixel graphic input

Reduce the **complexity of the interface and the assumptions** the package makes about input data, actions, and environment.



Simple Interface



Complex Interface

Carnegie Mellon University

Assumes only a 2D pixel graphic input

Design Principle for Design for Reuse: **Simple, Well-Documented Interfaces**

Reduce the **complexity of the interface and the assumptions** the package makes about input data, actions, and environment.

- Fewer assumptions ⇒ larger domain of **possible reuse contexts**
- **Explicitly Document Assumptions** underlying the **semantics** of the interface (e.g., color space of the image being RGB or LAB, image should be pre-filtered, or image should be small)

Reduce the **complexity of the interface and the assumptions** the package makes about input data, actions, and environment.

Design Principle for Design for Reuse: **Simple, Well-Documented Interfaces**

```
// DOM code to write an XML document to a specified output stream.
private static final void writeDoc(Document doc, OutputStream out) throws IOException {
    try {
        Transformer t = TransformerFactory.newInstance().newTransformer();
        t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
        t.transform(new DOMSource(doc), new StreamResult(out));
    }
    catch(TransformerException e) {
        throw new AssertionError(e); // Can't happen!
    }
}
```

Requires Setter Call

Few Arguments

Requires Exception Handling

How well does `TransformerFactory` support the principle?

See https://docs.oracle.com/javase/8/docs/api/javax/xml/transform/TransformerFactory.html

Carnegie Mellon University

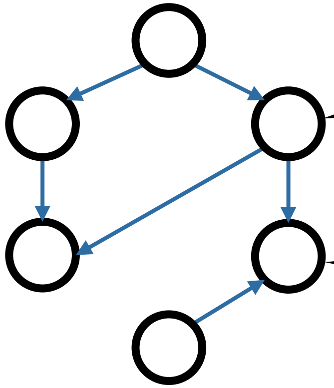
Effectiveness might depend on contrast and noise level

Does not depend on other *filters*

Design Principle for Design for Reuse: **Loose Coupling**

Coupling is the **degree of interdependence between** modules

Each module should **depend on as few** components as possible.
Dependencies should be **explicit** and **minimize** assumptions.

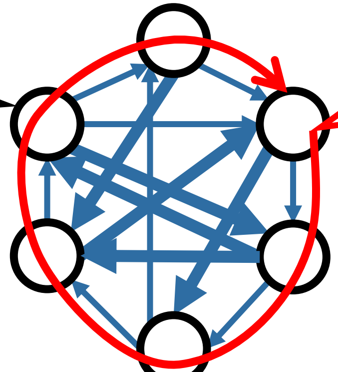


Medium Reusability

High Reusability

Low Coupling

Low Reusability



Avoid **cyclic dependencies**

High Coupling

Legend

Module

Dependency

Communicate
Generate
Evaluate

Carnegie Mellon University

Effectiveness might depend on contrast and noise level

Does not depend on other *filters*

Design Principle for Design for Reuse: **Loose Coupling**

Coupling is the **degree of interdependence between** modules

Each module should **depend on as few** components as possible. Dependencies should be **explicit** and **minimize** assumptions.

- Modules with **fewer dependencies** are easier to reuse, because it's easier to integrate them into a new context
- **Cyclic dependencies** prevent individual reuse

Each module should **depend on as few** components as possible.
Dependencies should be **explicit** and **minimize** assumptions.

Design Principle for Design for Reuse:
Loose Coupling

Coupling is the **degree of interdependence between** modules

Implicit dependency added

```
public static TransformerFactory newInstance()
            throws TransformerFactoryConfigurationError {
        String className = "org.apache.xalan.processor.TransformerFactoryImpl";
        try {
            return (TransformerFactory) Class.forName(className).newInstance();
        } catch (Exception e) {
            throw new NoClassDefFoundError(className);
```

Depends on small interface Source & Result

Adds dependency to custom Exception classes

```
public abstract void transform(Source xmlSource,
Result outputTarget) throws TransformerException
```

How well does TransformerFactory support the principle?

See https://android.googlesource.com/platform/prebuilts/fullsdk/sources/android-29/+/refs/heads/androidx-recyclerview-recyclerview-selection-release/javax/xml/transform/TransformerFactory.java

Each *filter* does **only one thing**

Design Principle for Design for Reuse: **High Cohesion**

Cohesion is the degree to which elements **within a module** are functionally **related**

Elements within a module should **work together** to fulfill a **single, well-defined purpose.**

Noise Reduction Filter

Sharpening Filter

Small modules tend to be more reusable

Larger cohesive modules can be reusable too

High Cohesion

What if I want an **unsharpened** image?

Noise Reduction & Sharpen Filter

Complete Image Filter Library (1 TB)

What if I need **only one filter**?

Low Cohesion

Carnegie Mellon University

Design Principle for Design for Reuse: **High Cohesion**

Cohesion is the degree to which elements **within a module** are functionally **related**

Each *filter* does **only one thing**

Elements within a module should **work together** to fulfill a **single, well-defined purpose.**

- Reusing a module that has multiple purposes adds **unnecessary pseudo-dependencies**
- It is easier to **understand** a module with high cohesion

Elements within a module should **work together** to fulfill a **single, well-defined purpose.**

Design Principle for Design for Reuse: **High Cohesion**

Cohesion is the degree to which elements **within a module** are functionally **related**

All about creating XML Transformers

```
* A TransformerFactory instance can be used to create
* {@link javax.xml.transform.Transformer} and
* {@link javax.xml.transform.Templates} objects.
* The system property that determines which Factory implementation
* to create is named "javax.xml.transform.TransformerFactory"
* This property names a concrete subclass of the
* TransformerFactory abstract class.
* If the property is not defined, a platform default is be used.
```

How well does `TransformerFactory` support the principle?

See https://android.googlesource.com/platform/prebuilts/fullsdk/sources/android-29/+/refs/heads/androidx-recyclerview-recyclerview-selection-release/javax/xml/transform/TransformerFactory.java

Carnegie Mellon University

Quiz on Design Principles

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.**

Carnegie Mellon University

How to
Design Complex Domain Logic To Be Reusable?

Designing Large-scale Software Systems - Design For Reuse

Carnegie Mellon University

Barely Reusable

Reusable In Few Contexts

Reusable In Many Contexts

How Reusable Are These Modules? Why?

Barely Reusable

Flight Storage Manager

(De-)Serializes Flight Data Structures to/from JSON files

Reusable In Few Contexts

JSON Mailbox

Sends a JSON file from one server to another server

Reusable In Few Contexts

Flight Tax Calculator

Given Flight Details, Calculates Air Transportation Taxes

Reusable In Many Contexts

Calculator

Calculates a given formular on numeric data tables

Carnegie Mellon University

Module Categories (“Blood Types”)

A-Module (Application Module) A	**T-Module** (Technology Module) T	**AT-Module** (Application + Technology) AT	**0-Module** (Independent Module) 0
Software that knows about the **application domain** and **business logic** (e.g., obstacle detection, tax calculation,	Software that knows about a **concrete technology** (e.g., MongoDB, JDBC, OpenGL, OpenCV, Windows API, …)	**Mixed** application logic and technology	**No dependency** on technology or application domain. implements an abstract concept, e.g., a dictionary or a state model

Carnegie Mellon University

A
Application Modules

T
Technology Modules

AT
Application & Technology Modules

0
Zero Modules

Which Blood Types Do The Modules Have?

Barely Reusable

Flight Storage Manager AT

(De-)Serializes Flight Data Structures to/from JSON files

T **JSON Mailbox**

Reusable In Few Contexts

Sends a JSON file from one server to another server

Reusable In Few Contexts

Flight Tax Calculator A

Given Flight Details, Calculates Air Transportation Taxes

0 **Calculator**

Reusable In Many Contexts

Calculates a given formular on numeric data tables

Carnegie Mellon University

A Application Modules

T Technology Modules

AT Application & Technology Modules

0 Zero Modules

Design Principle for Design for Reuse: **Minimize AT-Modules, Maximize 0-Modules**

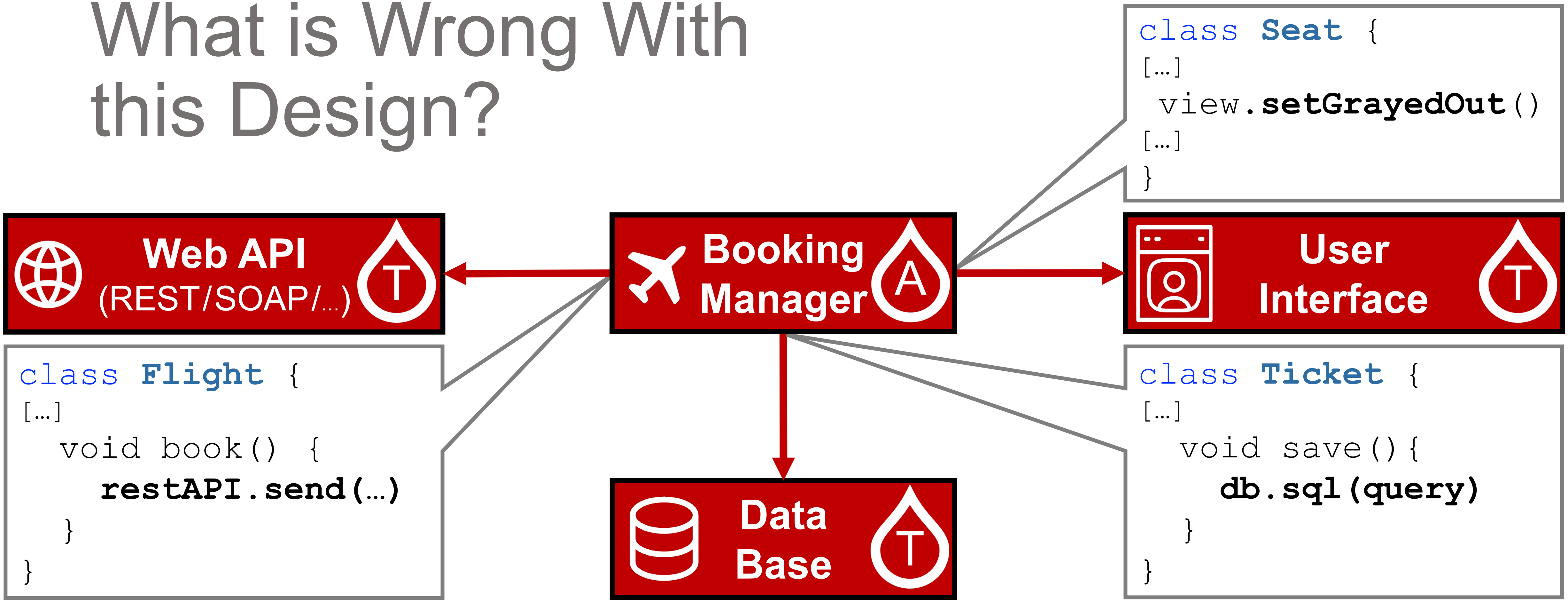
- **Assumptions on Technologies limit Reusability** to software that uses this technologies. Software with different technology cannot reuse the module
- **Assumptions on the Application Domain limit Reusability** to software in that domain. Different domains cannot reuse the module.
- **Therefore:** **Separate Technological Concerns from Application Concerns** to **avoid AT-Modules** or minimize their size

Refinement of Single Responsibility Principle

Carnegie Mellon University

A Application Modules | T Technology Modules | AT Application & Technology Modules | 0 Zero Modules

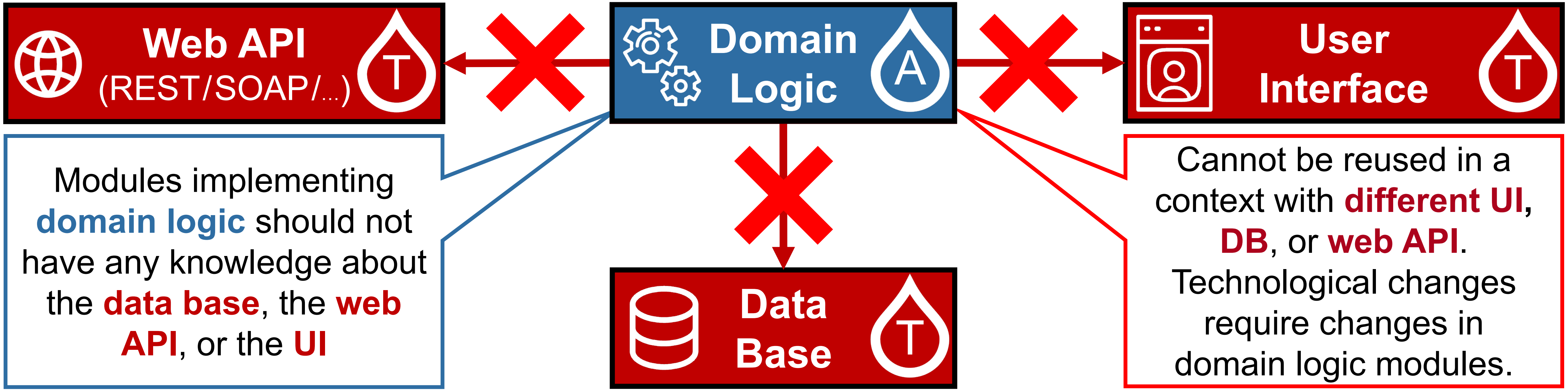
What is Wrong With this Design?



Carnegie Mellon University

Refinement of Dependency Inversion Principle

Design Principle for Design for Reuse: **Avoid Dependencies from Large & Complex A Modules to T Modules**



Carnegie Mellon University

The **UI**, **Data Base**, and **Web Interfaces** are **implementation details** that should **not drive the architecture** of the application

Design Recipe for Design for Reuse: **Invert Dependencies to the Web, UI, and Data Base**

Observer, MVC, Model-View-Presenter

Web API (REST/SOAP/...) T

Domain Logic A

User Interface T

Treat the web as a **plug-in** into the system.

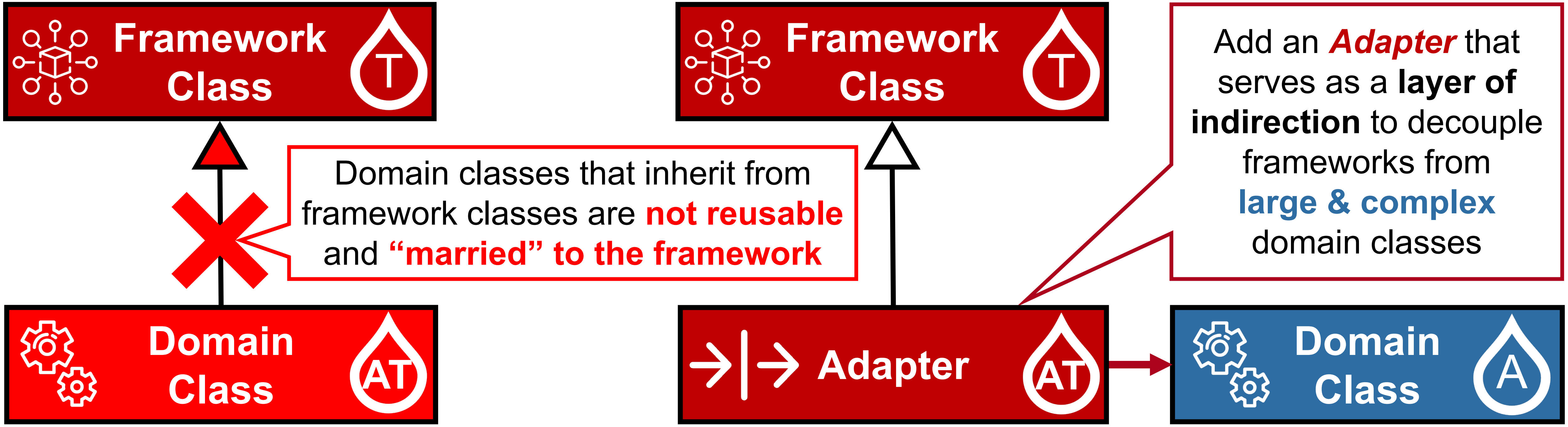
Data Base T

Object Relational Mapping
Dependency Injection

Carnegie Mellon University

Frameworks are **implementation details** that should **not drive the architecture** of the application

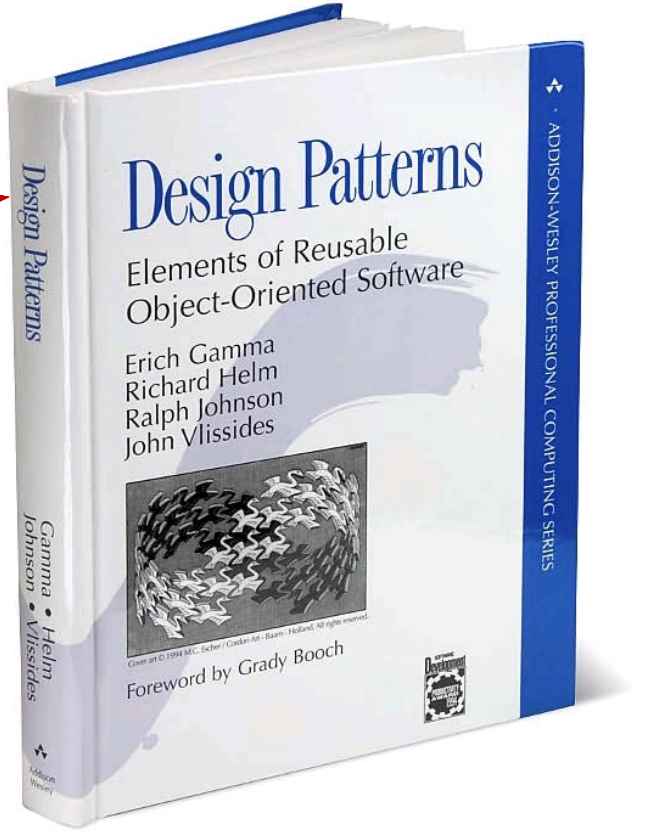
Design Recipe for Design for Reuse: **Reduce Coupling to Frameworks**



Patterns that Support Reusability

- Decorator
- Abstract Factory
- Composite
- Observer
- Template Method

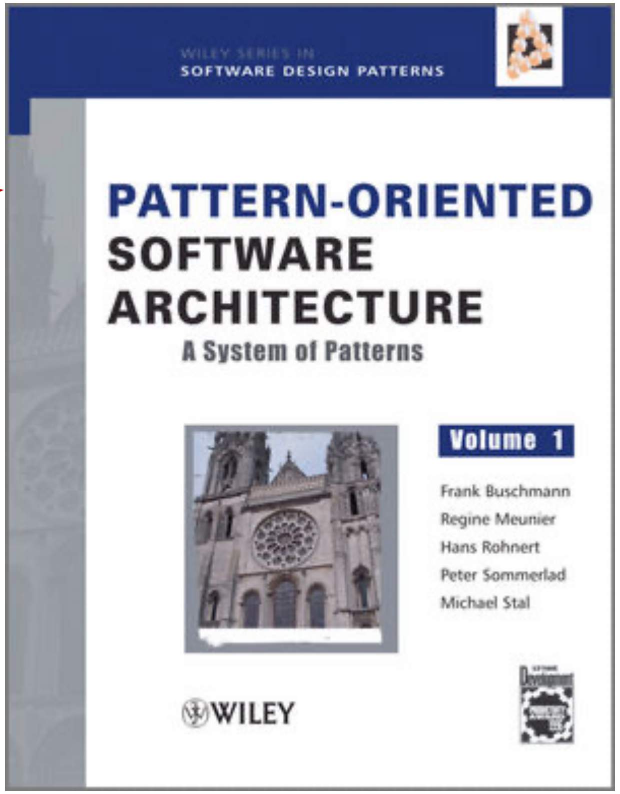
Read about them here!



Architectural Styles that Support Reusability

- Layers
- Pipes & Filters
- Publish-Subscribe

Read about them here!



Carnegie Mellon University

How to **Evaluate Reusability?**

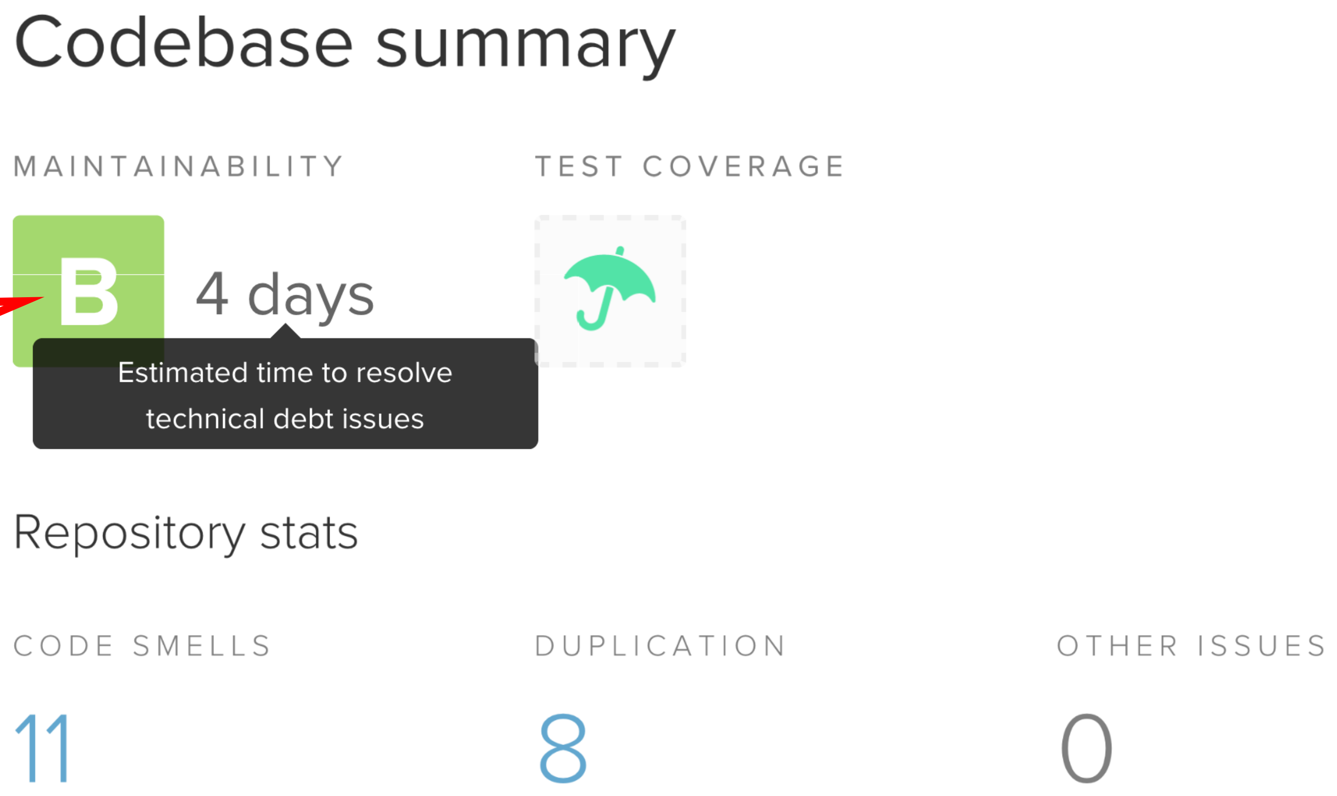
Designing Large-scale Software Systems - Design For Reuse

How useful are these metrics?

Tool: CodeClimate

Tools for Metric Analysis

High-level aggregated metrics are rarely useful



Carnegie Mellon University

How useful are these metrics?

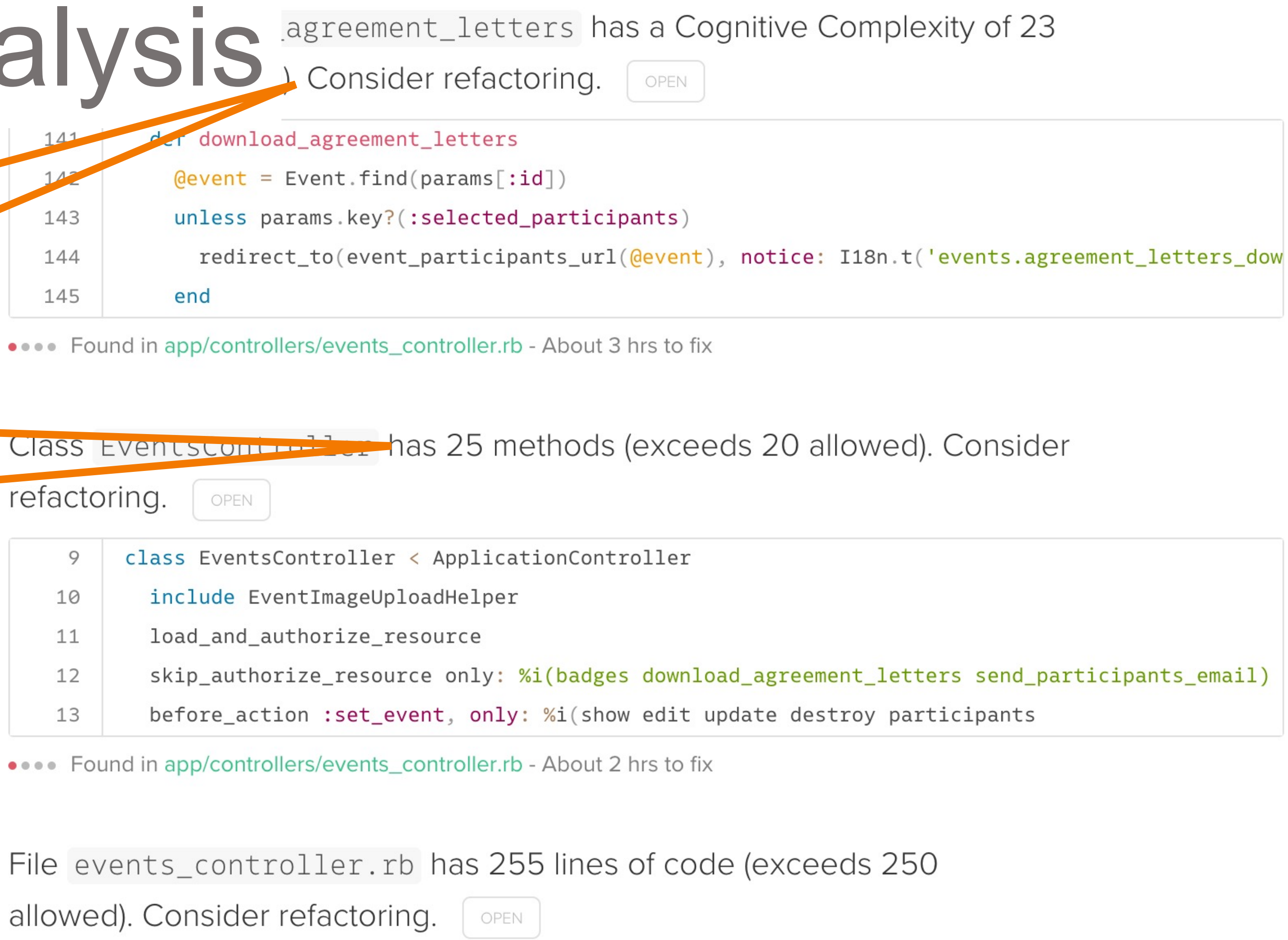
Tool: Codefactor

Tools for Metric Analysis

Ambiguous metrics often feel confusing to developers

General size metrics don’t reflect the design adequately

Syntactic code metrics are useful only if **not followed blindly** & the **semantics of metrics are clear**



Identify Reuse Scenarios

- Think of **different systems** for which a module would be **useful**
- Identify **ways in which they differ** from the current system (e.g., different domain, technology, …)
- Describe what amount of **effort of adaptation** would be reasonable based on the number of expected reuse clients

Carnegie Mellon University

Evaluate Reuse Scenarios

- **Identify assumptions** that the implementation makes about its context
- Check whether the assumptions **hold for all reuse scenarios**
- Identify potential **challenges** of reusing the system in the new context

Carnegie Mellon University

How does **Reusability Relate to Other Quality Attributes?**

Designing Large-scale Software Systems - Design For Reuse

Carnegie Mellon University

Connection To **Changeability**!

- Separation of software in A-modules and T-modules **increases changeability**!
- **Technology will change** over time (e.g., CORBA → REST, EJB → Spring, IBM Db2 → MongoDB)
- **Localizing the changes** required to adapt new technology to T-modules makes it easier to modernize the software

Connection To **Changeability**!

- **Loose Coupling**
- **High Cohesion**
- **Simple Interfaces**

} **support Changeability**

Carnegie Mellon University

Connection To **Testability**

- Reusable modules are **easier testable**
  - **Low coupling** increases testability
  - **Simple Interfaces** increases testability
  - **Modules that do not depend on the web, DB, or UI** are easier to test

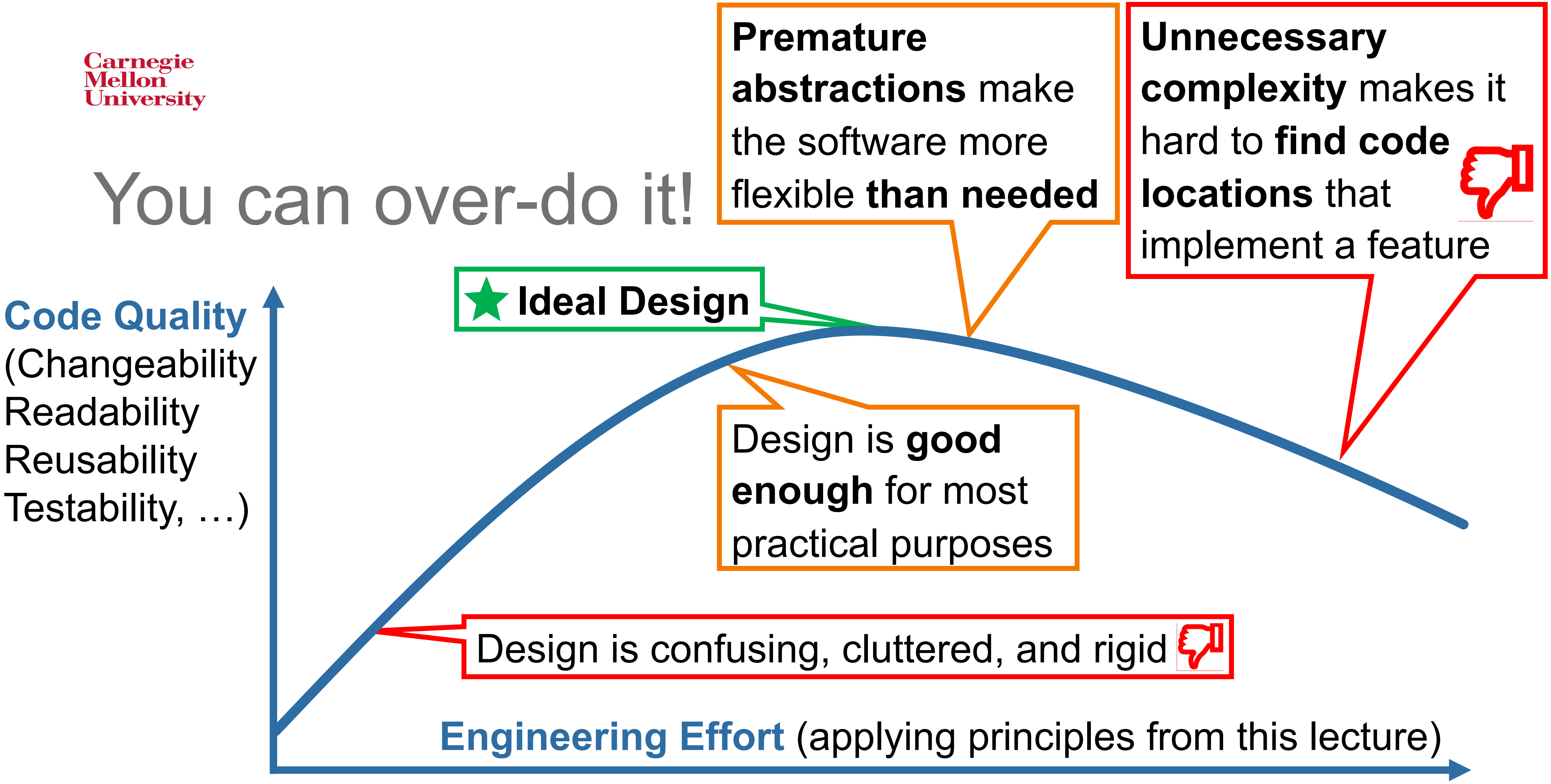
Carnegie Mellon University

Over-Engineering

- Solution is **more flexible** or sophisticated **than needed**
- **Premature abstractions** make it hard to **find code locations** that implement a feature
- Wasted **time** caused by **perfectionism**
- Unnecessary **complexity**

Carnegie Mellon University

You can over-do it!



Carnegie Mellon University

Connection To **Performance**

- More reusable designs can, in some cases, be **slightly slower**
- However: Unless you are building **embedded systems** with very **strict performance requirements**, the difference will be **minimal**

Carnegie Mellon University

Connection To **Interoperability**

- Reusability & Interoperability are **largely orthogonal**

Carnegie Mellon University

Please Complete the Exit Ticket in Canvas!

Question 1 **1 pts**

Please briefly summarize one or more key message from today's lecture (1~2 sentences).

Question 2 **1 pts**

Describe one **design principle** for Design for Reuse and describe at least one **example** that you encountered in your previous software development experience that **adhered** to this principle **or violated** it.

Question 3 **1 pts**

Please leave any questions that you have about today's materials and things that are still unclear or confusing to you (if none, simply write N/A).

Carnegie Mellon University

Summary

- Reusability supports understandability
- Loose Coupling, High Cohesion, and Simple Interfaces support Reusability
- Minimize AT-Modules, Maximize 0-Modules
- Avoid Dependencies from Large & Complex A Modules to T Modules
- Reduce Coupling to Frameworks