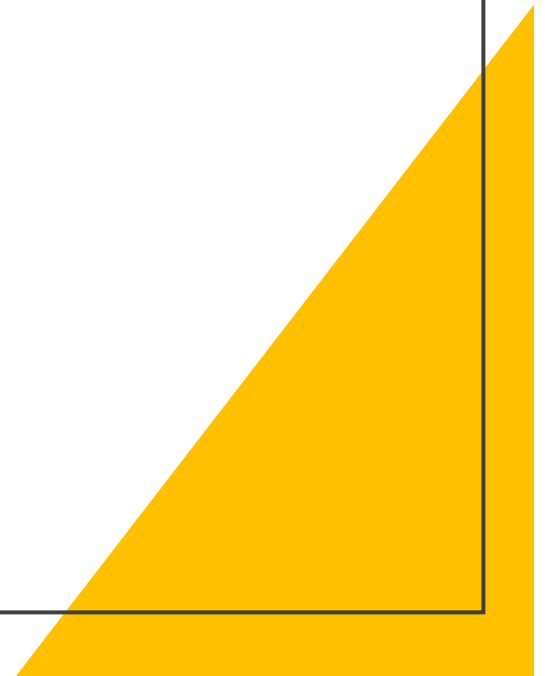


API Design

CSCI 420: Software Engineering

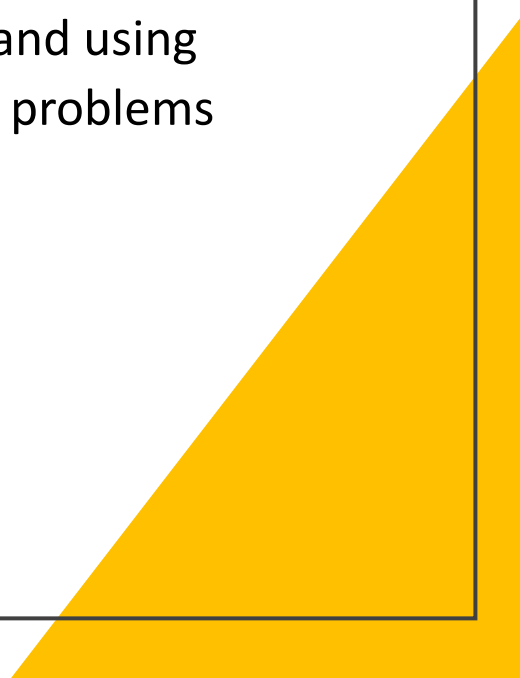
Millersville University



API

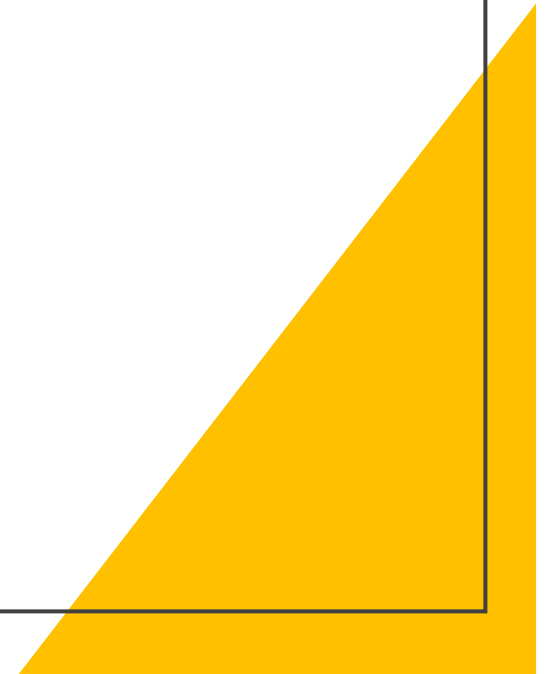
- Application Programming Interface
- Source Code Interface
 - For Library or Operating System
 - Provides services to a Program
- At its base, comparable to a header file
 - But, much more complete

Why is API Design Important?


- Company View
 - Can be asset – big user investment in learning and using
 - Bad design can be source of long-term support problems
 - Once used, it's tough to change
 - Especially if there are several users
 - Public APIs – One chance to get it right
- 
- A large yellow triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

Characteristics of Good APIs

- Easy to learn
- Easy to use even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to audience



Designing an API

- Gather requirements
 - Don't gather solutions
 - Extract true requirements
 - Collect specific scenarios where it will be used
 - Create short specification
 - Consult with users to see whether it works
 - Flesh it out over time
 - Hints:
 - Write plugins/use examples before fully designed and implemented
 - Expect it to evolve
- 
- A large yellow triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

Broad Issues to Consider in Design

1. Interface Principles
 - The classes, methods, parameters, names
 2. Resource Management
 - How is memory, other resources dealt with
 3. Error Handling
 - What errors are caught and what is done
- Information Hiding
 - How much detail is exposed
 - Impacts all three of the above

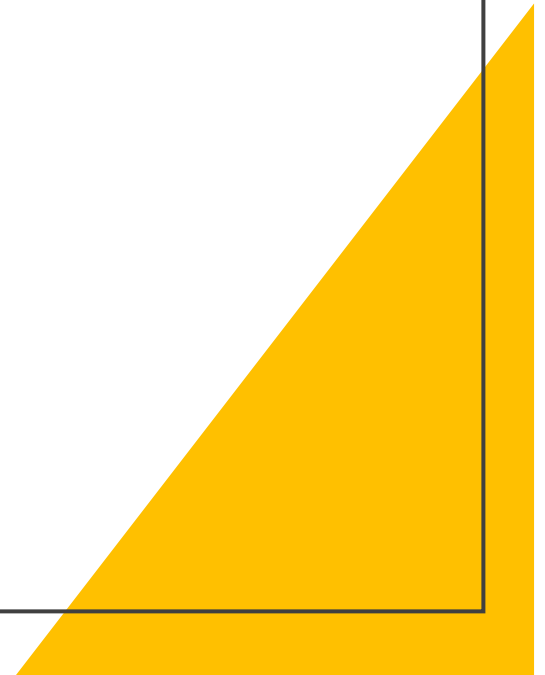
1

Interface Principles



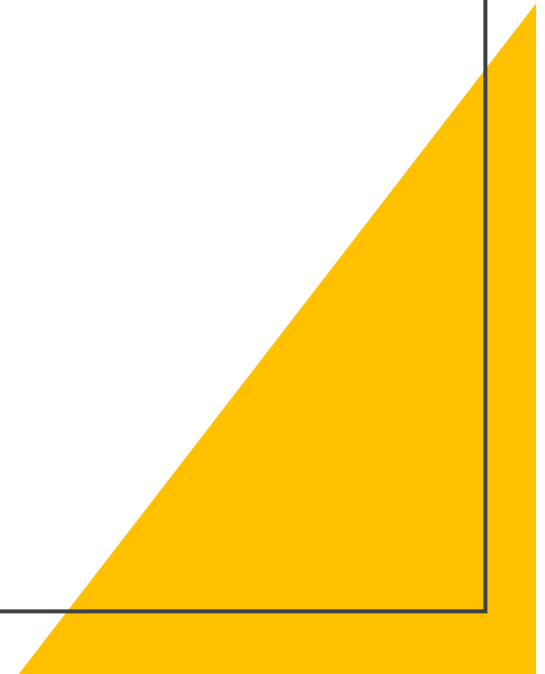
Interface Principles

- Simple
- General
- Regular
- Predictable
- Robust
- Adaptable



Simple

- Users must understand!
- Do one thing and do it well
 - Functionality should be easy to explain
- As small as possible, but never smaller
 - Conceptual weight more important than providing all functionality
 - Avoid long parameter lists
- Choose small set of orthogonal primitives
 - Don't provide 3 ways to do the same thing




General

- Implementation can change, API can't
- Hide Information!
 - Don't let implementation detail leak into API
 - Minimize accessibility (e.g. private classes and members)
 - Implementation details can confuse users
- Be aware of what is implementation
 - Don't overspecify behavior of modules
 - Tuning parameters are suspect

Regular

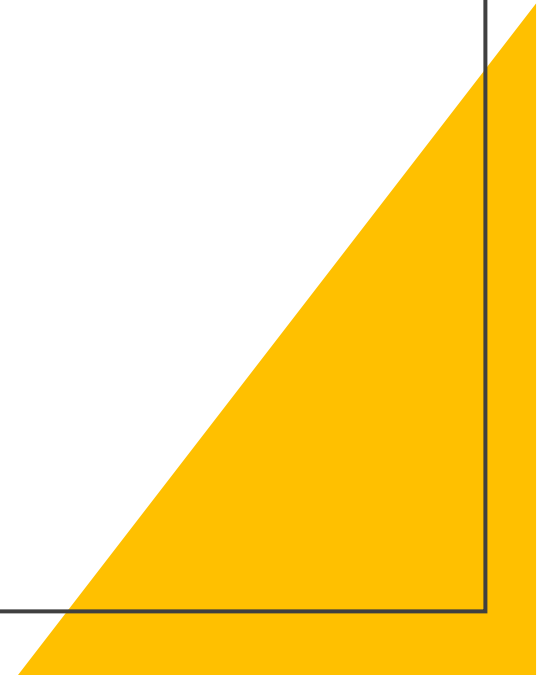
- Do the same thing the same way everywhere
 - Related things should be achieved by related means
 - Consistent parameter ordering, required inputs
 - Functionality (return types, errors, resource management)
- Names matter
 - Self explanatory
 - Consistent across API
 - Same word means same thing in API
 - Same naming style used
 - Consistent with related interfaces outside the API

Predictable

- Don't violate the principle of Least Astonishment
 - User should not be surprised by behavior
 - Even if this costs performance
 - Don't reach behind the user's back
 - Accessing and modifying global/static variables
 - Secret files or information written
 - Try to minimize use of other interfaces
 - Make as self-contained as possible
 - Be explicit about external services required
 - Document!
 - Every class, method, interface, constructor, exception
 - Mention states (in stateful applications)
- 
- A large yellow triangle is positioned in the bottom right corner of the slide, pointing towards the top right.

Robust

- Able to deal with unexpected input
- Error Handling (see later)



Adaptable

- API can be extended, but never shortened
 - Heavily used APIs likely will be extended
- Information Hiding
 - Implementation details should not affect API

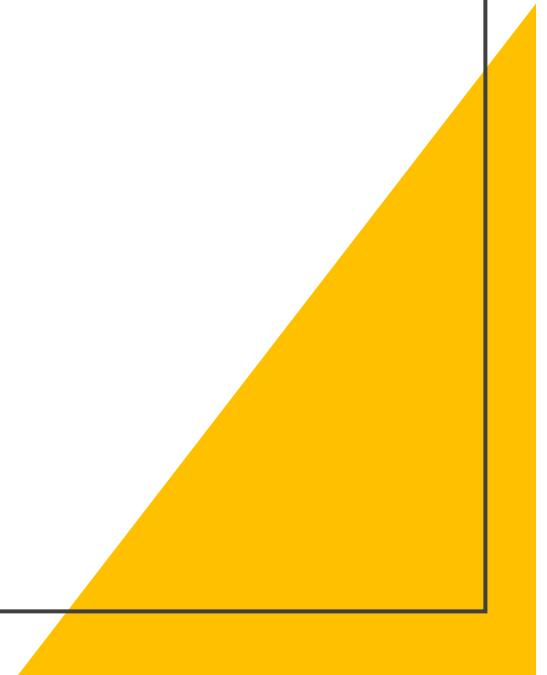
2

Resource
Management



Resource Management

- Determine which side is responsible for
 - Initialization
 - Maintaining state
 - Sharing and copying
 - Cleaning up
- Various resources
 - Memory
 - Files
 - Global variables



Resource Management

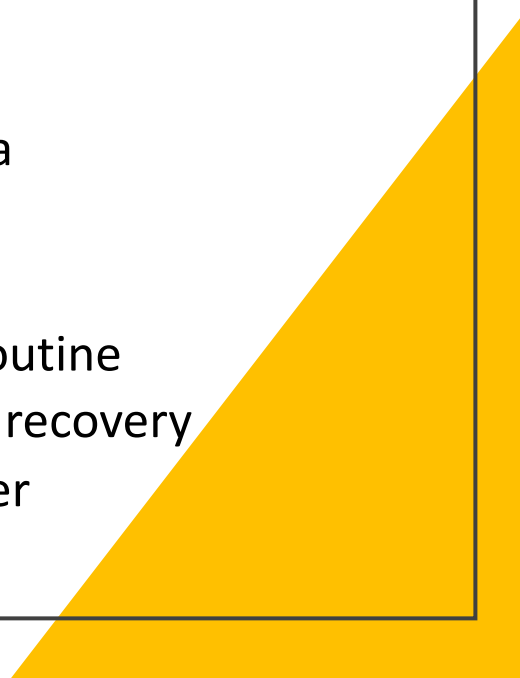
- Generally, free resources where they were allocated
- Return references or copies?
 - Can have huge performance and ease of use impact
- Multi-threaded code makes this especially critical
 - Reentrant: works regardless of number of simultaneous executions
 - Avoid using anything (globals, static locals, other modifications) that others could also use
 - Locks can be important

3

Error
Handling

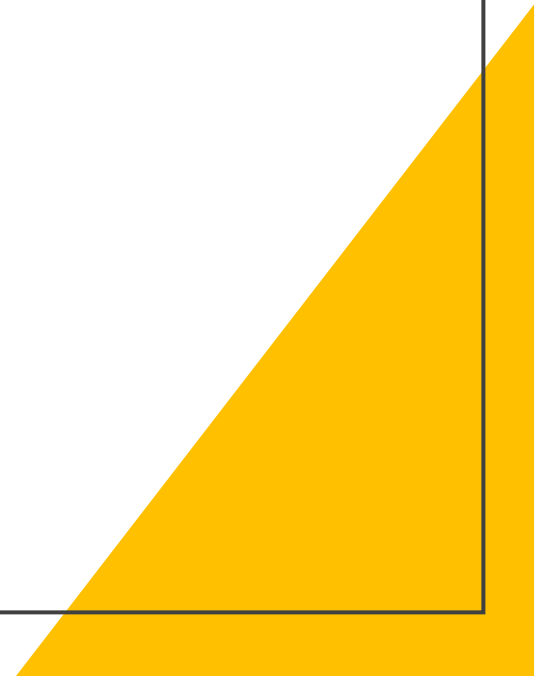


Error Handling

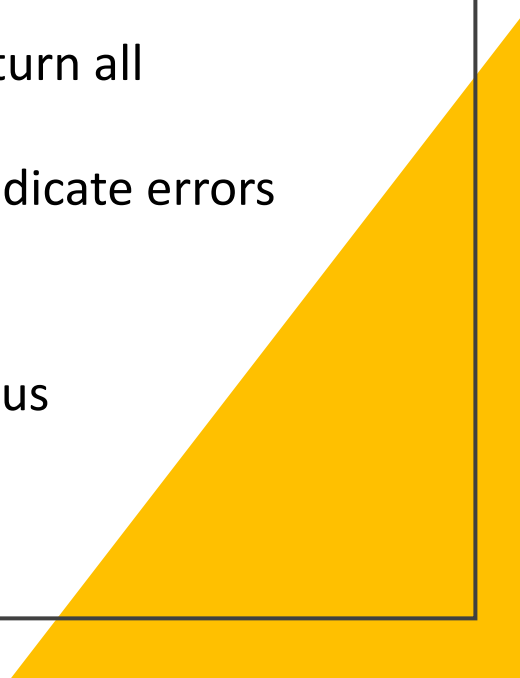
- Catch errors, don't ignore them
 - “Print message and fail” is not always good
 - Especially in APIs
 - Need to allow programs to recover or save data
 - Detect at low level, but handle at high level
 - Generally, error should be handled by calling routine
 - The callee can leave things in a “nice” state for recovery
 - Keep things usable in case the caller can recover
- 
- A yellow triangular graphic is located in the bottom right corner of the slide, pointing towards the top right.

Fail Fast

- Report as soon as an error occurs
- Sometimes even at compile time!
 - Use of static types, generics



Error Management

- Return Values
 - Should be in form the calling function can use
 - Return as much useful information as possible
 - Sentinel values only work if function cannot return all possible values of that type
 - Define pairs, or return another parameter to indicate errors
 - Use Error “wrapper function” if Needed
 - Consistent way of marking, reporting error status
 - Encourages use
 - But, can add complexity
- 
- A yellow triangular graphic is located in the bottom right corner of the slide, pointing towards the top right.

Exceptions

- Generally indicate a programming error
- Programming construct
 - Set exception value (e.g. as return)
 - Other program operation when exception thrown
 - Exceptions usually in global registry
- Include information about failure
 - For repair and debugging
- Exceptions should generally be unchecked
 - Automatically process globally, rather than require explicit checks over and over

Exceptions

- Only use in truly exceptional situations
 - Never use as a control structure
 - The modern GOTO
- Never use exceptions for expected return values
 - e.g. Invalid file name passed to library is “common”, not an exception