University of
South Australia

INFS 2044

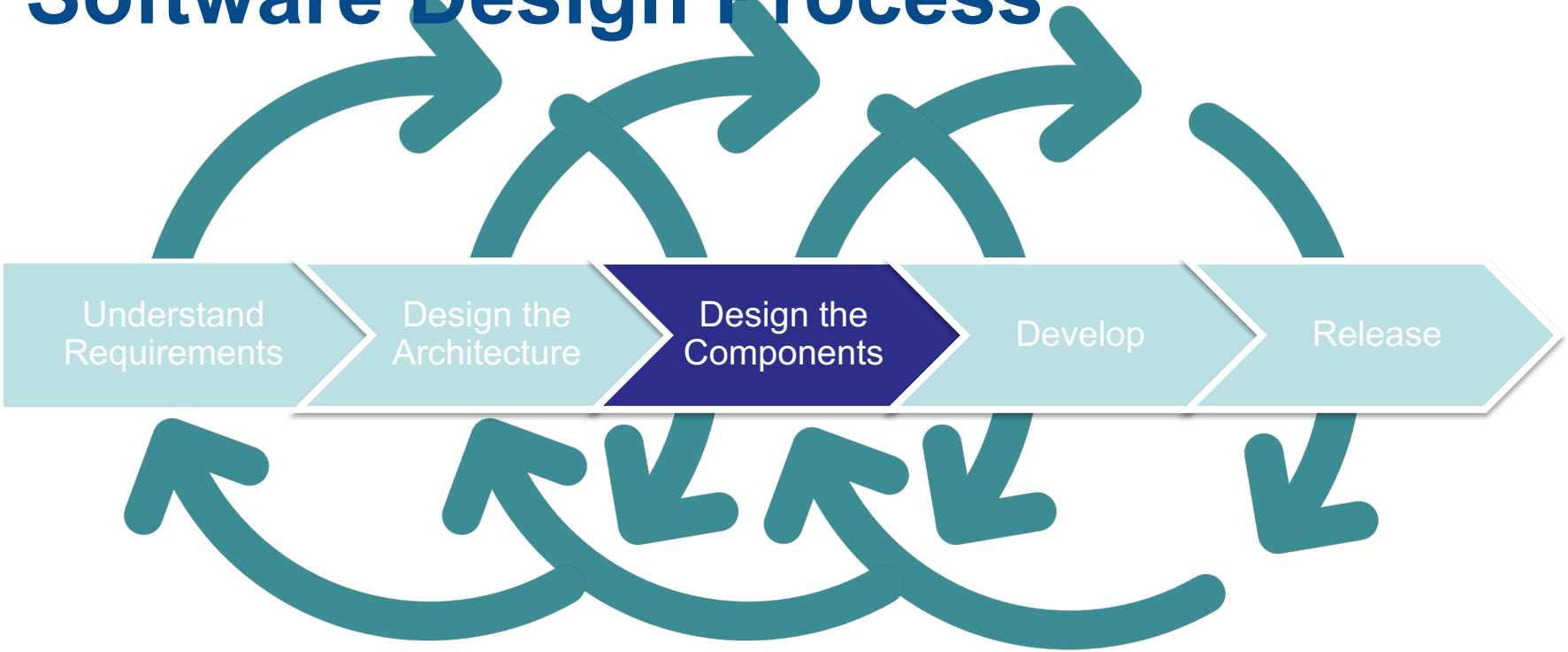
Week 3

Modules and Boundaries

Learning Objectives

- Understand the concepts of abstraction and information hiding (CO2)
- Design interfaces based on abstraction and information hiding (CO4)

Software Design Process

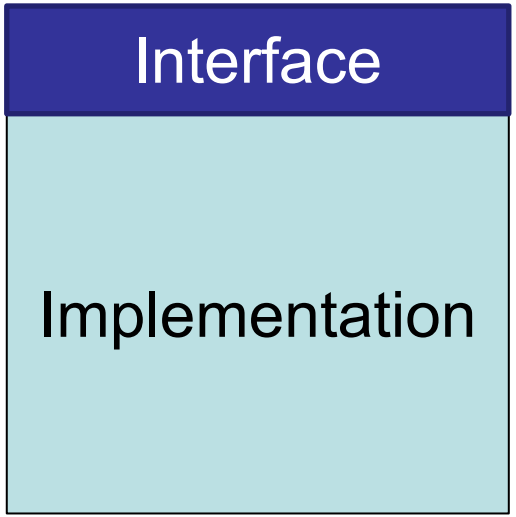


Architecture Design Recap

- Divide system into components that are relatively independent
- Ideal: each component completely independent of the others
  - System complexity = complexity of worst component
- In reality, components are not completely independent
  - Some components must invoke facilities in other components
  - Design decisions in one component must sometimes be known to other components
  - Can't change one component without understanding parts of other components

Interfaces

- Divide each component into two parts:
- **Interface**: anything about the component that must be known to other components
- ***Implementation***: code that carries out the promises made by the interface
- Interface should be much simpler than the implementation



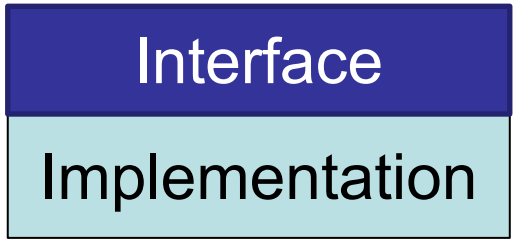
What is in an Interface?

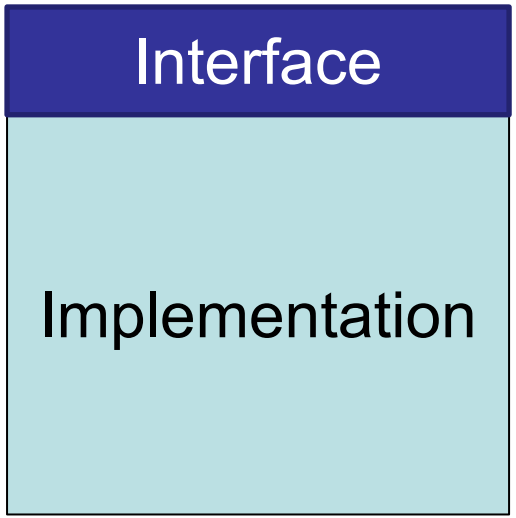
- Formal aspects (in the code)
  - Operation signatures, public variables, etc.
- Informal aspects:
  - Overall behaviour, side effects, constraints on usage, etc.
  - Usually described with comments and documentation

Abstraction

- A simplified view of something that omits unimportant details
- Interface: abstraction of a component
- Goal: define simple abstractions that provide rich functionality
- Apply this principle to components, classes, etc

Shallow vs Deep





Example: File I/O

Python (Deep)

```
open(file, mode='r', buffering=-1, encoding=None,
        errors=None, newline=None,
        closefd=True, opener=None)
```

Java (Shallow)

```
FileInputStream fileStram =
                new FileInputStream(fileName);
BufferedInputStream bufferedStream =
                new BufferedInputStream(fileStream);
ObjectInputStream objectStream =
                new ObjectInputStream(bufferedStream);
```

Signs of Shallow Components

- Complex interface
- Little functionality
- Shallow components don't hide much information
- Every module and operation introduces complexity with its interface

- Goal: get a lot of functionality for that complexity

Size vs Depth

- Small classes taken to the extreme
  - Each class adds the least possible amount of functionality to existing classes
  - Bad example: Java libraries
- Size does not matter that much
  - The most important thing is depth: the power of the abstraction
- More important to have a **simple interface** than a simple implementation

Red Flag: Repetition

The same piece of code appears over and over again:

you have not found the right abstractions

Information Hiding

- **Each component should encapsulate certain knowledge or design decisions**
  - The knowledge/design decisions are only known to the one component
  - The interface does not reflect this information (much)
- Benefits
  - Simpler interface (deeper component)
  - Can modify the implementation without impacting other components
- This is the single most important idea in software design

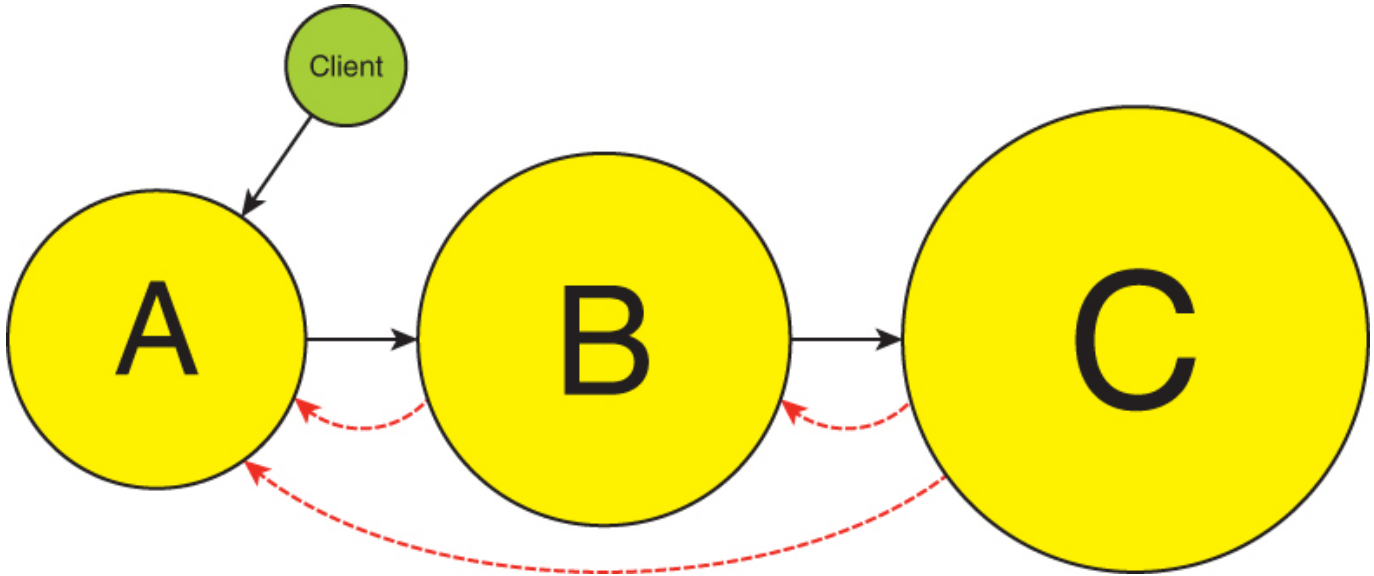
Information Leakage

- Opposite of information hiding
- Implementation details exposed
- Other modules depend on them
- Anything in the interface is leaked
- Back-door leakage: not visible in the interface

Leakage Examples

- How to store information in a B-tree, and how to access it efficiently.
- How to identify the physical disk block corresponding to each logical block within a file.
- How to implement the TCP network protocol.
- How to schedule threads on a multi-core processor.
- How to parse JSON documents.

Temporal Decomposition = Leakage



Classes can Leak

```
class ShoppingCart:

    def __init__():
        self.__items = dict()

    def getItems():
        return self.__items

    def addItem(item, qty):
        self.__items[item] = qty
```

```
cart = ShoppingCart()
cart.addItem("milk", 3)

items = cart.getItems()
items["bread"] = 1
```

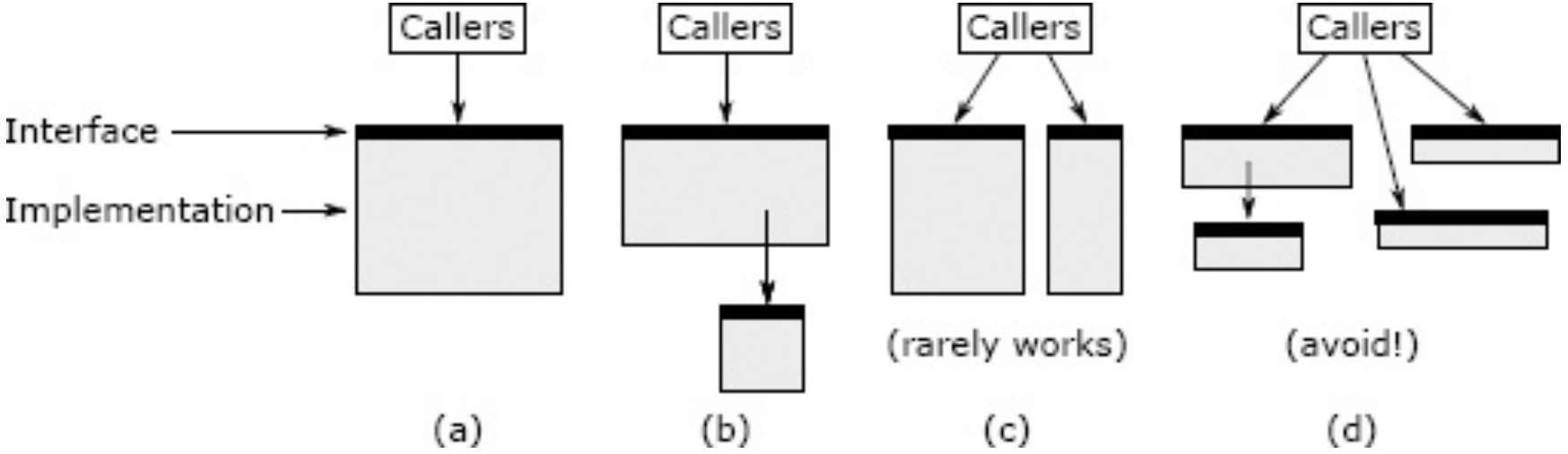
Avoiding Leakage: Questions to Ask

- What is the unique value provided by this component? (something this component does, but no other components)
- What is the key knowledge that the component uses to provide that value?
- What is the least possible amount of that knowledge that must be exposed through the interface?

Resolving Leakage

- Look for a way to bring all the information together in one place
  - Creates opportunities for better information hiding
  - Bring together if it will simplify the interface
  - Bring together to eliminate duplication
  - Separate general-purpose and special-purpose code
- Each operation should do one thing and do it completely

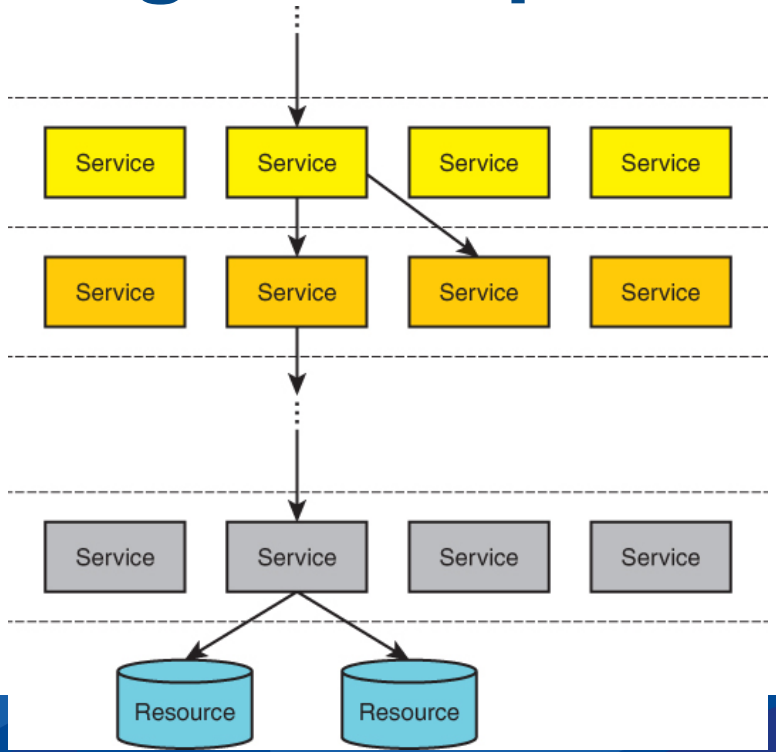
Together or Split?



Red Flags: Conjoined Code

- If you cannot understand the implementation of one operation without also understanding the implementation of another.
- If two pieces of code are physically separated, but each can only be understood by looking at the other.

Layered Design Recap



New Layer = New Abstraction

- Each layer's abstraction should be different from the layer above it and the layer below it.
  - Red flag: pass-through operations
- Decide *what is important*, design the interface around that
  - Focus on the things that are done most frequently

Red Flag: Pass-Through Operations

- A pass-through operation is one that does nothing except pass its arguments to an operation in another component, usually with the same signature as the pass-through operation.

- This indicates that there is not a clean division of responsibility between the components.

Designing Interfaces

- Technique #1: if a particular task is invoked repeatedly
  - design an API around that task
  - even better, do it automatically, without having to be invoked
- Technique #2: if a collection of tasks are not identical
  - look for common features shared by all of them; design APIs for the common features
  - Design APIs for infrequently-used features so that you don't need to be aware of them when using the common features

Java IO – Bad Example

```
FileInputStream fileStram =
        new FileInputStream(fileName);
BufferedInputStream bufferedStream =
        new BufferedInputStream(fileStream);
ObjectInputStream objectStream =
        new ObjectInputStream(bufferedStream);
```

Unix IO – Good Example

- UNIX emphasized commonality across devices:
  - Devices have names in the file system: special device files
- All devices have same basic access structure: open, read, write, seek, close
- Device-specific operations with one additional kernel call:
  - int result = ioctl(int fd, int request, void* inBuffer, int inputSize, void* outBuffer, int outputSize);

General vs Specific

- "Should I implement extra features beyond those that I need today?”
  - Design facilities that are general-purpose when possible (don't get carried away)
  - Don't create a lot of specific features that aren't needed now; you can always add them later.
  - When you discover that new features or a more general architecture are needed, do it right away: don't hack around it.

Pull Complexity Down

- Simple APIs are more important than a simple implementation
- Component writers should embrace suffering
  - Take on hard problems
  - Solve completely
  - Make solution easy for others to use
  - Let a few component developers suffer, rather than thousands of users

How to?

- Make 2 designs and compare
  - Pick one and write some code
  - Watch for red flags
  - Revise code
- Take advantage of code reviews

Summary

- Each component should encapsulate certain knowledge or design decisions
- Finding the correct abstraction is key
- Watch for red flags and information leakage

Activities this Week

- Read the required readings
- Participate in Workshop 3
- Complete Quiz 3

University of
South Australia