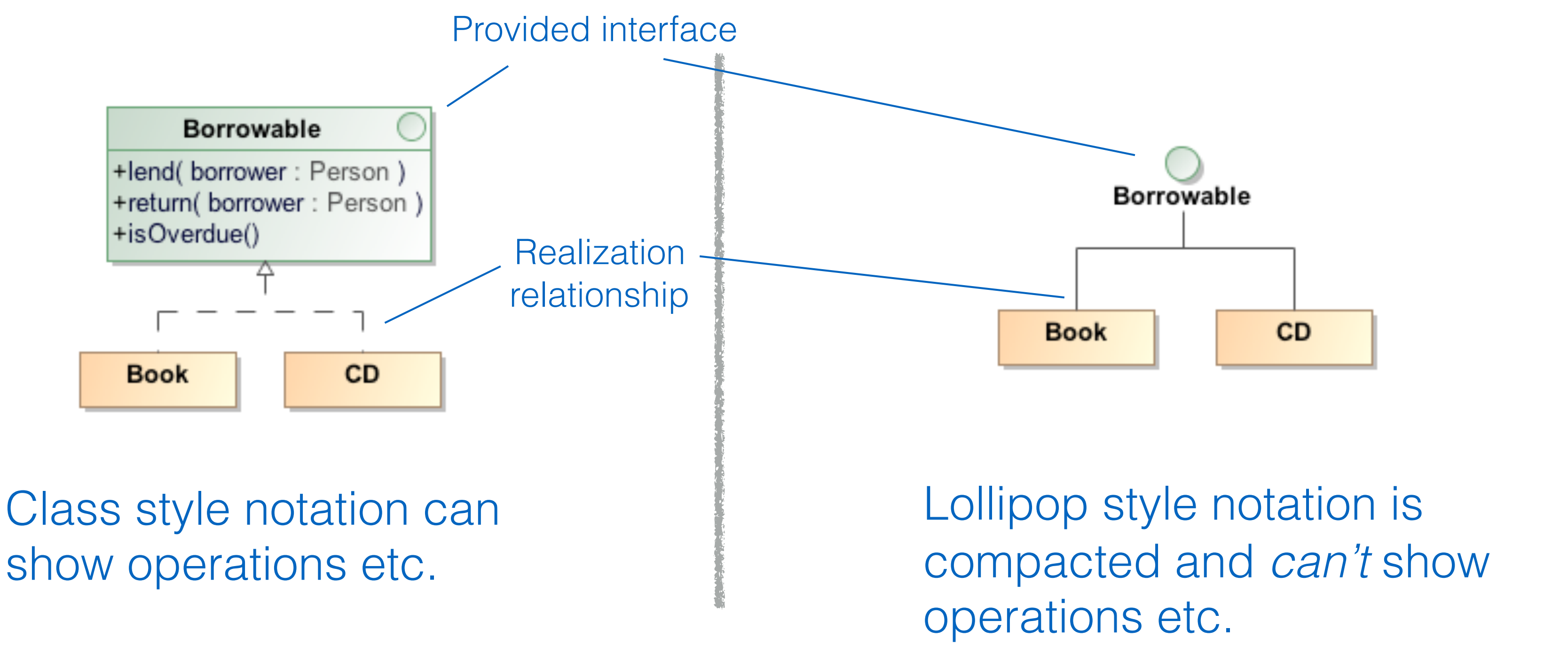
Design - interfaces & components

What is an interface?

- An interface specifies a named set of public features - it separates the *specification* of functionality (the interface) from its *implementation* in a realizing classifier
- An interface defines a contract that all realizing classifiers *must* conform to - this is *design by contract*

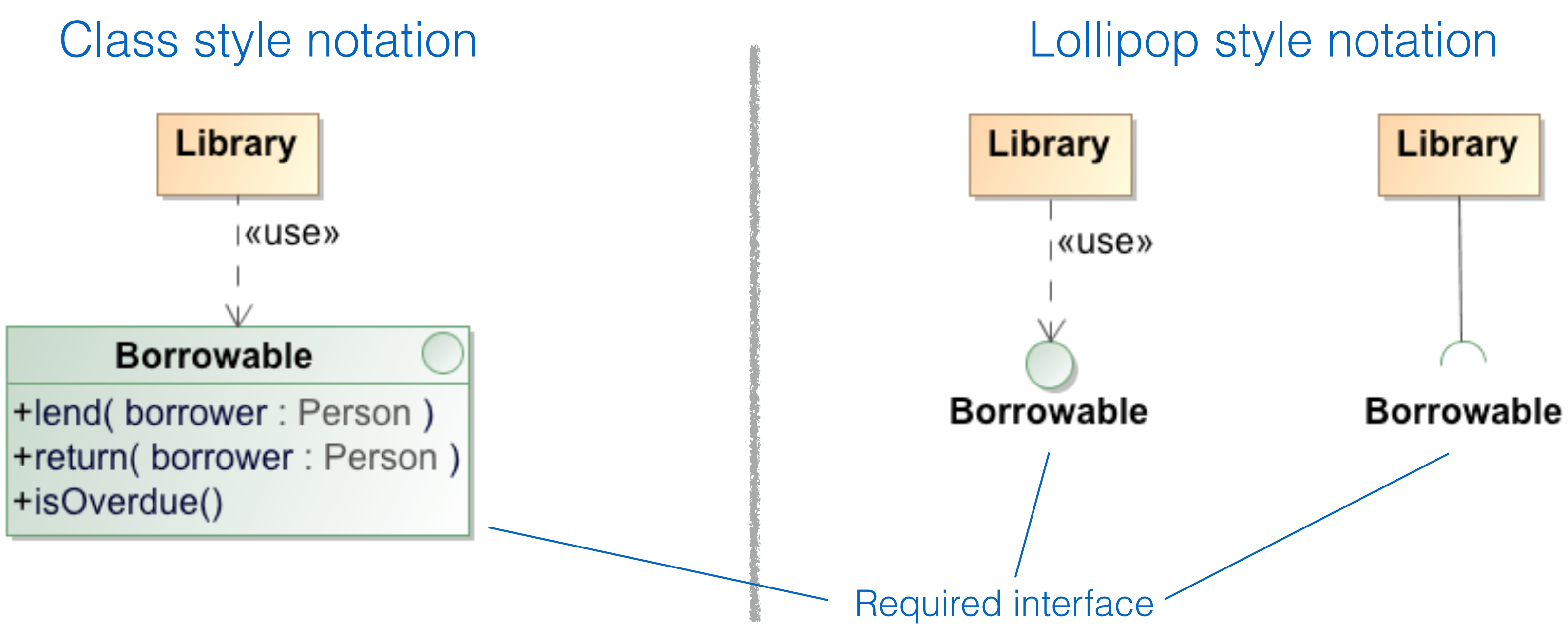
Interface feature	Realizing classifier
operation	Must have an operation with the same signature and semantics
attribute	Must have public operations to set and get the value of the attribute. The realizing classifier is *not required* to actually have the attribute specified by the interface, but it must behave as though it has
association	Must have an association to the target classifier. If an interface specifies an association to another interface, then the implementing classifiers of these interfaces must have an association between them
constraint	Must support the constraint
stereotype	Has the stereotype
tagged value	Has the tagged value
protocol	Must realize the protocol

Provided interface syntax



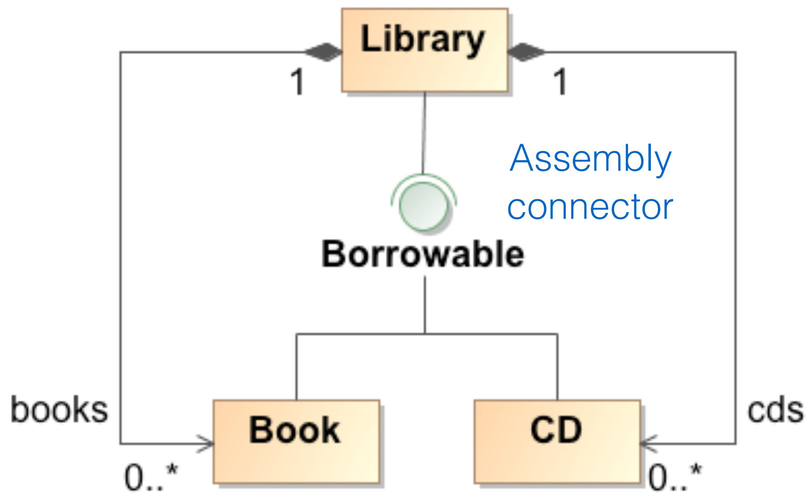
- A provided interface indicates that a classifier implements the services defined in the interface

Required interface syntax



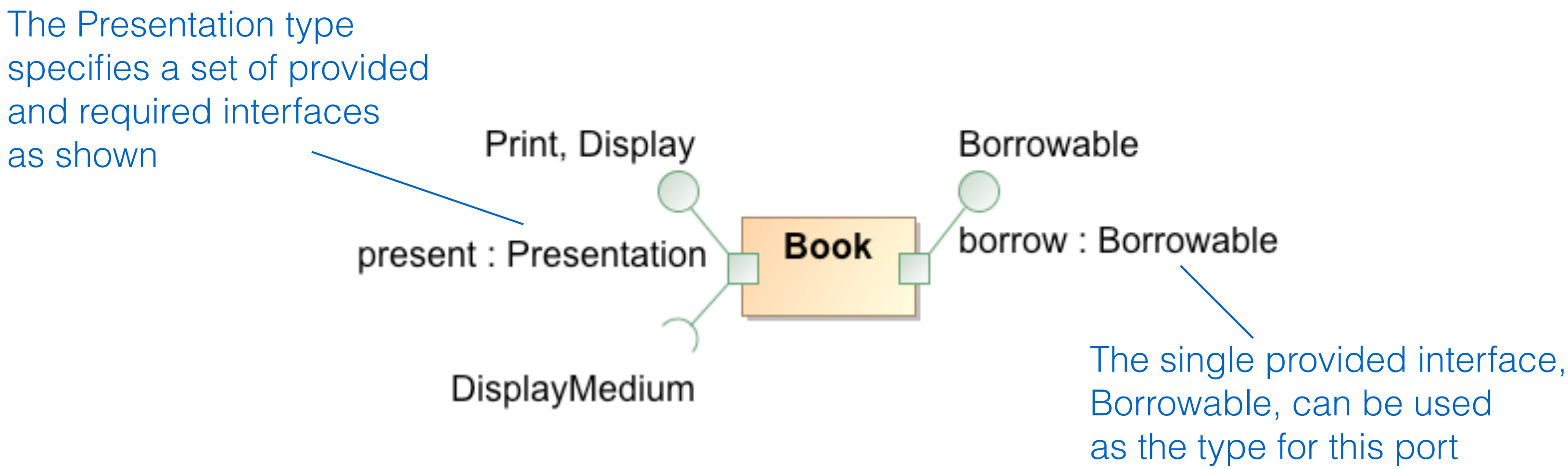
- A required interface indicates that a classifier uses the services defined by the interface

Assembly connectors



- You can connect provided and required interfaces using an assembly connector
- This is a convenient “plug and socket” style of syntax

Ports



- A port specifies an (optionally named) interaction point between a classifier and its environment. It is a semantically cohesive set of provided and required interfaces

- Each port has a type that is based on its provided and required interfaces. Ports of the same type have the same interfaces (of course!)

- If a port has a *single* provided interface, this can be the type of the port, and there is no need to create a new type

Interfaces and CBD

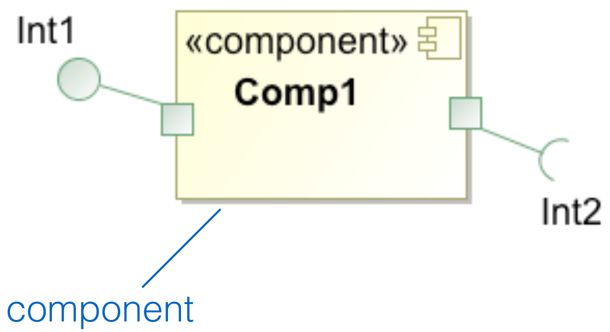
- Interfaces are the key to *Component Based Development* (CBD), which is about constructing software from replaceable, pluggable parts:
    - Plug – the provided interface
    - Socket – the required interface
- The concept of standard pluggable units is extensively used, e.g. electrical outlets and computer ports – USB, serial, parallel
- Interfaces define a contract. Classifiers that realize the interface, by definition, agree to abide by that contract and can be used interchangeably
- The idea is that you design to an interface, rather to any specific realization

What is a component?

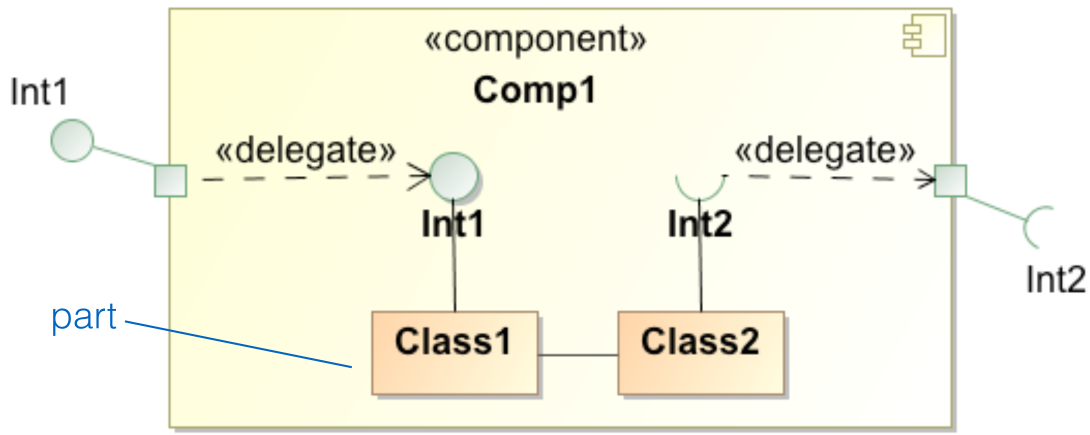
- The UML 2.5 specification states that, "A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment"
- It is a black-box whose external behavior is completely defined by its provided and required interfaces
- It may be substituted for by other components *provided* they support the same protocol
- Components can be:
  - Physical - can be directly instantiated at run-time, e.g. an Enterprise JavaBean (EJB)
  - Logical - a purely logical construct, e.g. a subsystem that is only instantiated indirectly by virtue of its parts being instantiated

Component syntax

Black box view



White box view



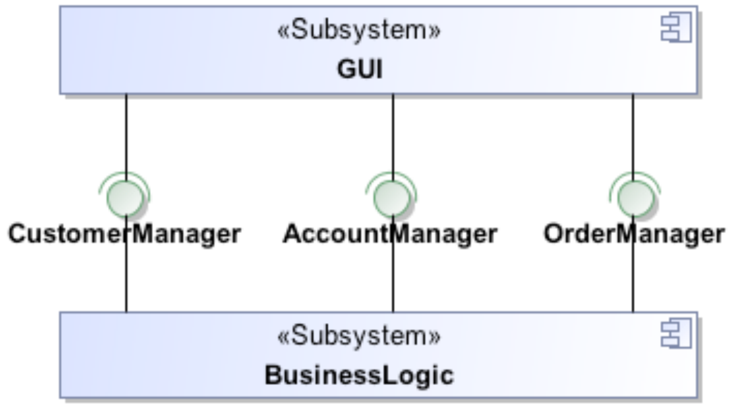
- Components may have provided and required interfaces, ports and internal structure comprising parts
- Provided and required interfaces usually delegate to internal parts
- You can show the parts nested inside the component icon or externally, connected to it by dependency relationships

Standard component stereotypes

Stereotype	Semantics
«BuildComponent»	A component that defines a set of things for organizational or system level development purposes (e.g. compilation)
«Entity»	A persistent information component representing a business concept
«Implement»	A component definition that is not intended to have a specification itself. Rather, it is an implementation for a separate «Specification» to which it has a dependency
«Specification» *	A classifier that specifies a domain of objects without defining the physical implementation of those objects. For example, a Component stereotyped by «Specification» only has provided and required interfaces - no realizing classifiers
«Process»	A transaction based component
«Service»	A stateless, functional component that computes a value
«Subsystem»	A unit of hierarchical decomposition for large systems

* «Specification» applies generally to all classifiers, not just components

Subsystems



- A subsystem is a component that acts as a unit of decomposition for a larger system
- It is a *logical* construct used to decompose a larger system into manageable chunks
- Subsystems can't be instantiated at run-time, but their contents can
- Interfaces connect subsystems together to create a system architecture

Finding interfaces and ports

- Challenge each association: Does the association have to be to a class, or can it be to an interface?
- Challenge each message send: Does the message send have to be to a class, or can it be to an interface?
- To find interfaces look for:
  - Repeating groups of operations
  - Groups of operations that might be useful elsewhere
  - Possibilities for future expansion - create a plug or socket
- Organize cohesive sets of provided and required interfaces into named ports
- Consider the dependencies between subsystems and mediate these by an assembly connector where possible

Designing with interfaces

- Design interfaces based on common sets of operations
- Design interfaces based on common roles. These roles may be between two classes or even within one class which interacts with itself. They may also be between subsystems
- Design interfaces for new plug-in features and algorithms
- The Façade Pattern - use interfaces can be used to create "seams" in a system:

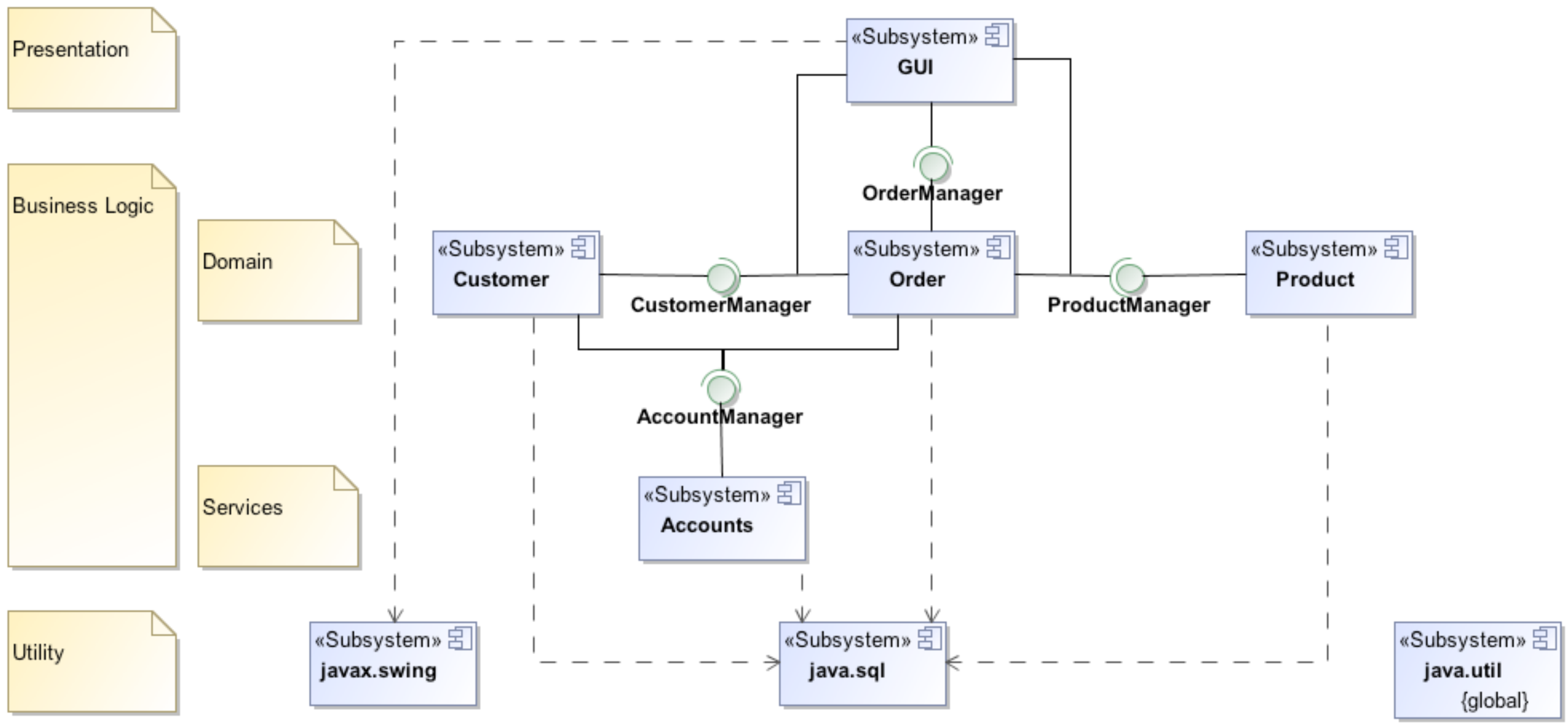
    1. Identify cohesive parts of the system and package these into a «subsystem»
    2. Define an interface to that subsystem

Interfaces allow information hiding and separation of concerns!

Architecture

- Subsystems and interfaces comprise the architecture of our model, and we need to organize them to create a coherent architectural picture
- On way is to apply the "layering" architectural pattern in which subsystems are arranged into layers:
  - Each layer contains design subsystems which are semantically cohesive, e.g. Presentation layer, Business logic layer, Utility layer
  - Dependencies between layers are very carefully managed
  - Dependencies go one way
  - Dependencies are mediated by interfaces

Example layered architecture



Using interfaces

- Pros:
  - When we design with classes, we are designing to specific implementations
  - When we design with interfaces, we are instead designing to contracts which may be realized by many different implementations (classes)
  - Designing to contracts frees our model from implementation dependencies and thereby increases its flexibility and extensibility
- Cons:
  - Interfaces can add flexibility to systems BUT flexibility may lead to complexity
  - Too many interfaces can make a system too flexible!
  - Too many interfaces can make a system hard to understand

Keep it simple!

Summary

- Interfaces specify a named set of public features that define a contract that classes and subsystems may realize
- Programming to interfaces rather than to classes reduces dependencies between the classes and subsystems in our model
- Programming to interfaces increases flexibility and extensibility but may add complexity
- Design subsystems and interfaces allow us to componentize our system and define an architecture