Software Engineering (Part 2)

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover these software engineering topics:
  - Requirements analysis
  - Design
  - Implementation
  - Debugging
  - Testing
  - Evaluation
  - Maintenance
  - Process models

Stages of SW dev

How to order the stages

Objectives

Software Engineering lectures:

Part 1	Requirements analysis Design (general)
Part 2	**Design (object-oriented)** **Implementation** **Debugging**
Part 3	Testing Evaluation
Part 4	Maintenance Process models

Agenda

- Requirements analysis
- **Design**
- Implementation
- Debugging
- Testing
- Evaluation
- Maintenance
- Process models

Design: OO Heuristic 1

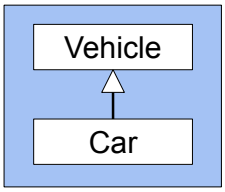
- Use **inheritance** to model "**is a**" (or "is a kind of")
- Use **composition** to model "**has a**"

- Examples…

Design: OO Heuristic 1

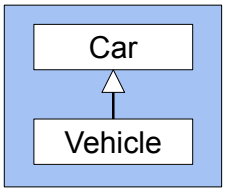
Which is proper?

1



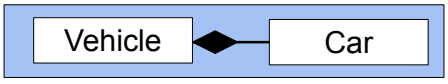
Car inherits from Vehicle

2



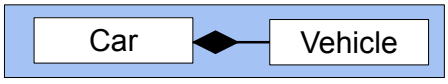
Vehicle inherits from Car

3



A Vehicle object is composed of a Car object

4

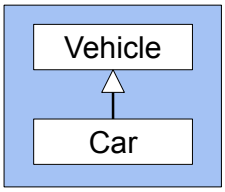


A Car object is composed of a Vehicle object

Design: OO Heuristic 1

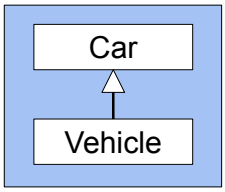
Which is proper?

1



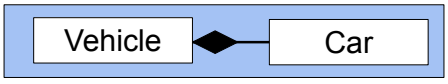
Car inherits from Vehicle

2



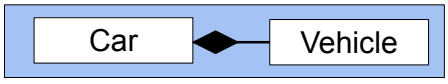
Vehicle inherits from Car

3



A Vehicle object is composed of a Car object

4

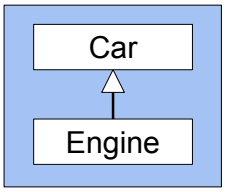


A Car object is composed of a Vehicle object

Design: OO Heuristic 1

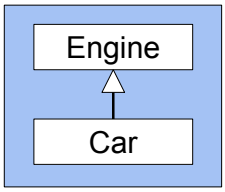
Which is proper?

1



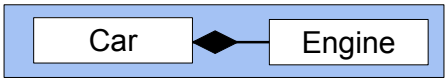
Engine inherits from Car

2



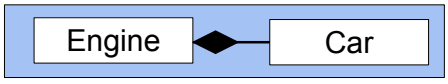
Car inherits from Engine

3



A Car object is composed of an Engine object

4

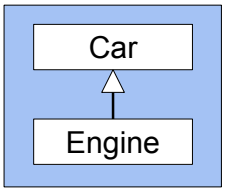


An Engine object is composed of a Car object

Design: OO Heuristic 1

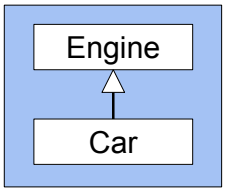
Which is proper?

1



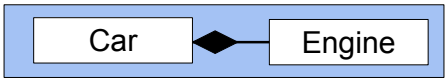
Engine inherits from Car

2



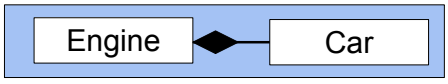
Car inherits from Engine

3



A Car object is composed of an Engine object

4

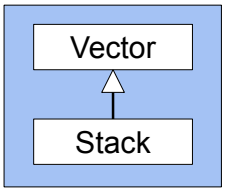


An Engine object is composed of a Car object

Design: OO Heuristic 1

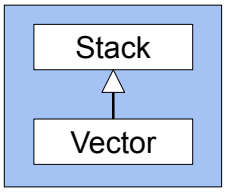
Which is proper?

1



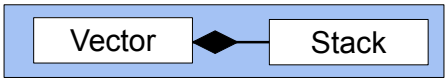
Stack inherits from Vector

2



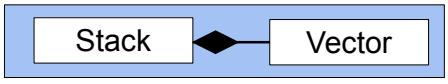
Vector inherits from Stack

3



A Vector object is composed of a Stack object

4

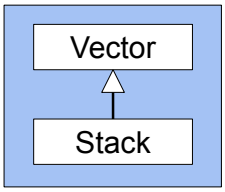


A Stack object is composed of a Vector object

Design: OO Heuristic 1

Which is proper?

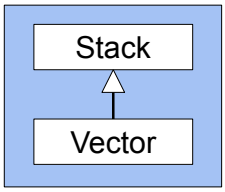
1



JDK does this

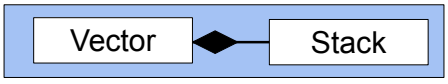
Stack inherits from Vector

2



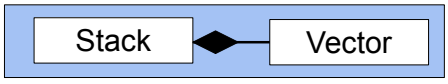
Vector inherits from Stack

3



A Vector object is composed of a Stack object

4



A Stack object is composed of a Vector object

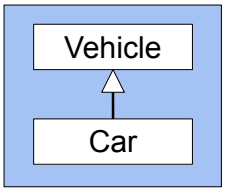
Design: OO Heuristic 2

- When designing inheritance hierarchies…
- Use the ***Liskov substitution principle***
  - Let p(t) be a property provable about objects t of type T. Then p(s) should be true for objects s of type S where S is a subtype of T
  - Informally: If class S inherits from class T, then an object of class S should be usable in place of an object of class T

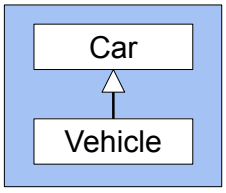
Barbara Liskov and Jeannette Wing.
"A behavioral notion of subtyping,"
ACM Transactions on Programming Languages and Systems,
Volume 16, Issue 6 (November 1994), pp. 1811 - 1841.

Design: OO Heuristic 2

- Use the Liskov sub principle (cont.)



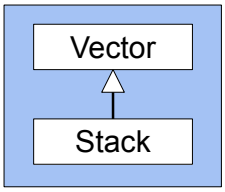
Suppose we have some code that uses a Vehicle object
Can we can replace the Vehicle object with a Car object and expect the code to work?
Yes!



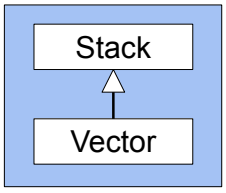
Suppose we have some code that uses a Car object
Can we can replace the Car object with a Vehicle object and expect the code to work? **No!**

Design: OO Heuristic 2

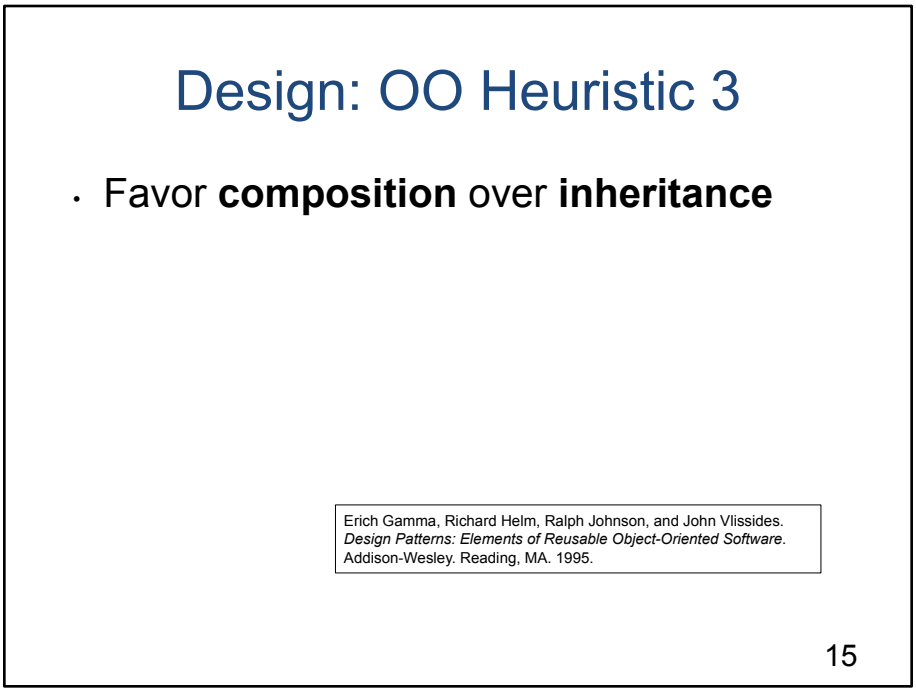
- Use the Liskov sub principle (cont.)



Suppose we have some code that uses a Vector object
Can we can replace the Vector object with a Stack object and expect the code to work? **No!**



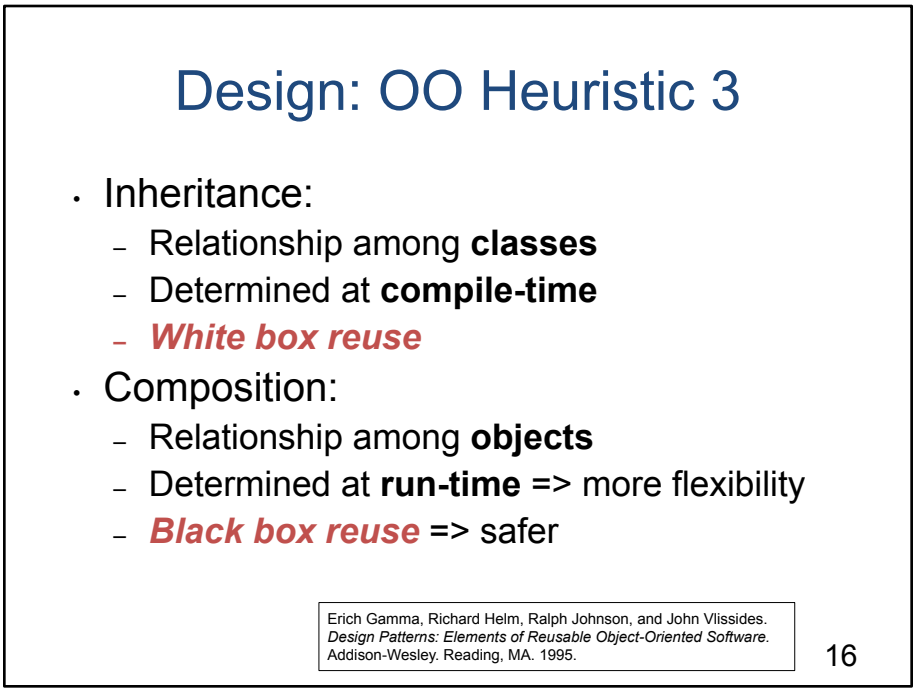
Suppose we have some code that uses a Stack object
Can we can replace the Stack object with a Vector object and expect the code to work? **No!**



Design: OO Heuristic 3

[see slide]

The point is made eloquently in the Design Patterns book
Stay tuned!



Design: OO Heuristic 3

Inheritance:
- Relationship among **classes**
- Determined at **compile-time**
- ***White box reuse***
  - Subclass object often/usually sees internals of superclass object
  - Change superclass => often/usually change subclass
  - Inheritance breaks encapsulation
  - Inheritance violates information hiding

Composition:
- Relationship among **objects**
- Determined at **run-time** => more flexibility
  - Example:
    - Compile-time: Any Car is composed of Engine
    - Run-time: This Car is composed of 4-cylinder Engine, or 6-cylinder Engine, or …
- ***Black box reuse*** => safer
  - Neither containing nor contained object sees internals of the other
  - Does not break encapsulation
  - Does not violate information hiding

So when you have a choice, choose composition over inheritance

I’m not a big fan of inheritance

I use it when appropriate

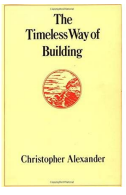
But I try to design broad & shallow inheritance hierarchies -- to minimize the encapsulation violations

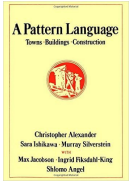
Design: OO Heuristic 4

- Use **OO design patterns**…

Aside: Architectural Patterns

- **Who**: Christopher Alexander
  - Architect, not computer scientist
- **When**: 1970s
- **Why**: Help people create rooms, buildings, towns, ...





Aside: Architectural Patterns

- Example: **Entrance Room**

> "Arriving in a building, or leaving it, you need a room to pass through, both inside the building and outside it. This is the entrance room."
>
> "At the main entrance to a building, make a light-filled room which marks the entrance and straddles the boundary between indoors and outdoors, covering some space outdoors and some space indoors. The outside part may be like an old-fashioned porch; the inside like a hall or sitting room."

Christopher Alexander et al.
A Pattern Language.
Oxford University Press. New York. 1977.

Aside: Architectural Patterns

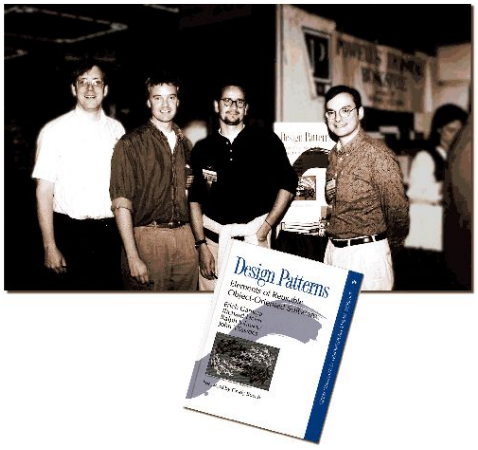
- Example: **Private Terrace on the Street**

> "The relationship of a house to a street is often confused: either the house opens entirely to the street and there is no privacy; or the house turns its back on the street, and communion with street life is lost."
>
> "Let the common rooms open onto a wide terrace or a porch which looks into the street. Raise the terrace slightly above street level and protect it with a low wall, which you can see over if you sit near it, but which prevents people on the street from looking into the common rooms."

Christopher Alexander et al.
A Pattern Language.
Oxford University Press. New York. 1977.

Design: OO Patterns

The Gang of Four



Ralph Johnson
Richard Helm
Erich Gamma
John Vlissides

Design: OO Patterns

Creational Patterns
- Abstract factory
- Builder
- Factory method
- Prototype
- Singleton

Structural Patterns
- Adapter
- **Bridge**
- **Composite**
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns
- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

Design: OO Pattern: Composite

- Example: ***Composite***

> "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1995.

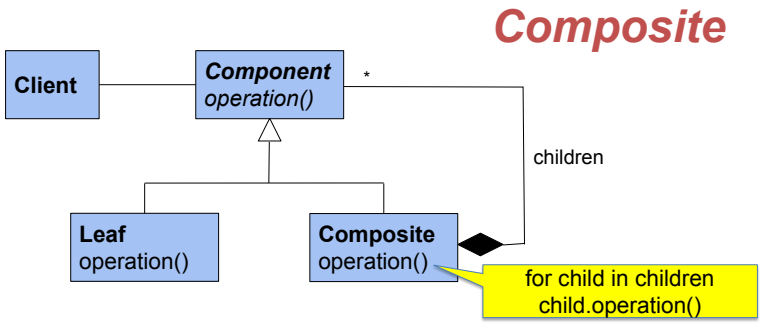
Design: OO Pattern: Composite

- Example: ***Composite*** (cont.)

> "Use the Composite pattern when:
>
> -- you want to represent part-whole hierarchies of objects
>
> -- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly."

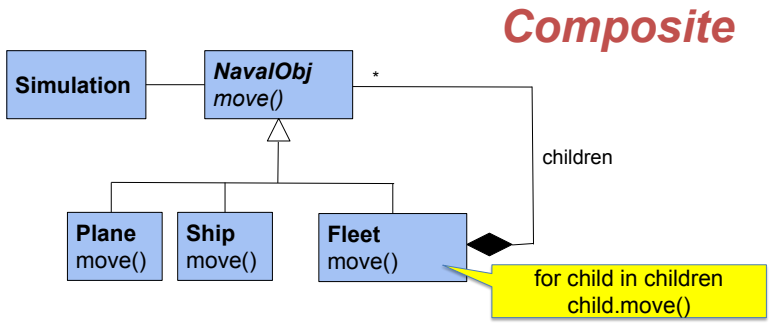
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

Design: OO Pattern: Composite



Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

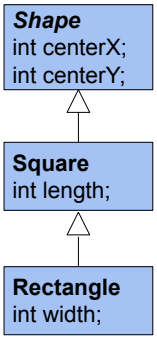
Design: OO Pattern: Composite



Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

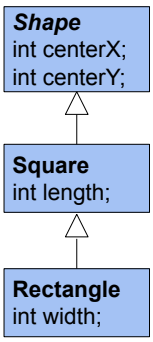
Design: OO Pattern: Bridge

- **Question**: Is this design proper?



Design: OO Pattern: Bridge

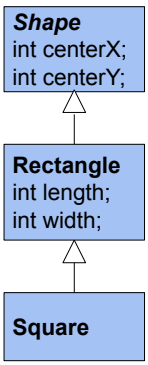
- **Question**: Is this design proper?



Answer:
No! Violates the Liskov substitution principle

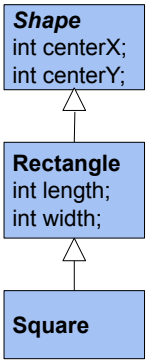
Design: OO Pattern: Bridge

- **Question**: Is this design proper?



Design: OO Pattern: Bridge

- **Question**: Is this design proper?



Answer:
Better! But each Square object has a field that's either unused or redundant -- or inconsistent!

Design: OO Pattern: Bridge

- Example: ***Bridge***

> "Decouple an abstraction from its implementation so that the two can vary independently."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1995.

Design: OO Pattern: Bridge

- Example: ***Bridge***

> "When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways. But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

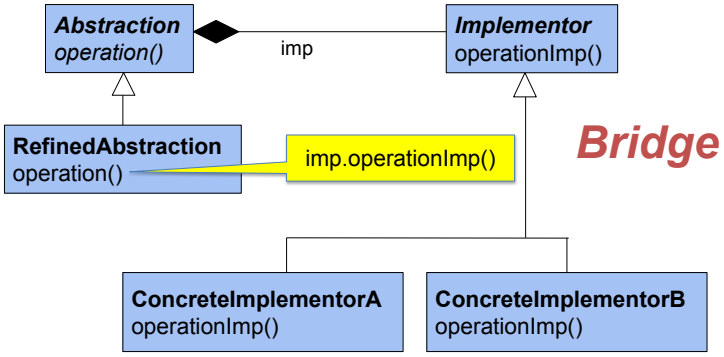
Design: OO Pattern: Bridge

- Example: ***Bridge*** (cont.)

> "Use the Bridge pattern when:
>
> -- you want to avoid a permanent binding between an abstraction and its implementation.
>
> -- both the abstractions and their implementations should be extensible by subclassing. In this class the Bridge pattern lets you combine the different abstractions and implementations and extend them independently."

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

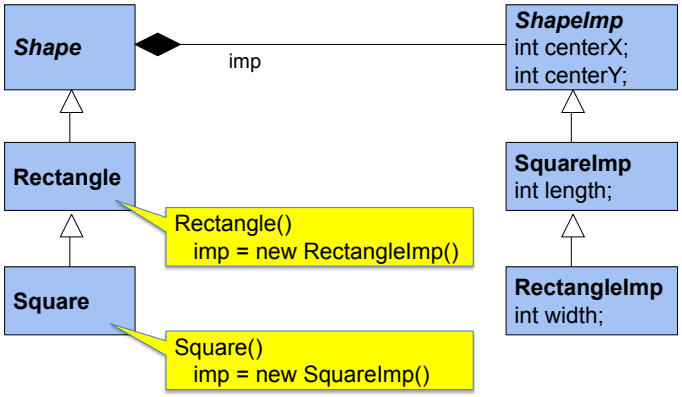
Design: OO Pattern: Bridge



Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

Design: OO Pattern: Bridge

- The proper design: ***Bridge***



Aside: OO Patterns in Other Fields

- **User interface** design patterns
  - See http://ui-patterns.com/
  - See http://www.welie.com/patterns/

Aside: OO Patterns in Other Fields

- **User interface** design patterns (cont.)
  - Example: Password strength meter

> "**Problem summary**: You want to make sure your users' passwords are sufficiently strong in order to prevent malicious attacks. **Solution**: A password's strength is measured according to predefined rules and is displayed using a horizontal scale next to the input field. If the password is weak then only a small portion of the horizontal bar is highlighted. The greater the strength of the password the more the horizontal bar is highlighted. The password strength is also appropriately indicated by coloring the bar in a color associative with good or bad: Green indicating a strong password and red indicating a weak password."

https://ui-patterns.com/patterns/PasswordStrengthMeter

Aside: OO Patterns in Other Fields

- **Pedagogical** design patterns!!!
  - See http://www.pedagogicalpatterns.org/
  - See *Pedagogical Patterns: Advice For Educators* (Bergin et al, editors)

You've designed your application. What should your next step be?

Agenda

- Requirements analysis
- Design
- **Implementation**
- Debugging
- Testing
- Evaluation
- Maintenance
- Process models

Implementation

- ***Implementation***
  - How should you code the system?
  - Coding the system yourself is only one option
  - Options…

Implementation: Buy

- **Option 1: Buy**
  - Buy an existing system
  - (pro) System is already tested and evaluated
  - (pro) System support provided by vendor
  - (con) System and system support cost money!!!

Implementation: Open Source

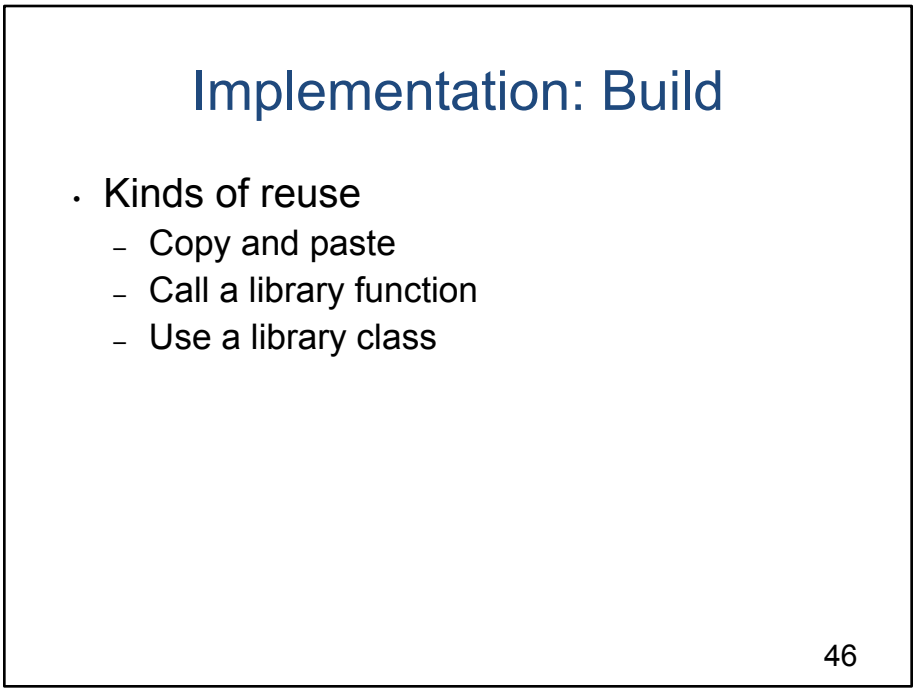
- **Option 2: Use open source**
  - Use an existing open source system
  - (pro) System (maybe) is already tested and evaluated
  - (pro) System is free
  - (con) (Maybe) must support the system yourself

Implementation: Open Source

- Open-source challenges
  - Finding the open-source code to be reused!
    - How to **categorize** reusable code?
    - How to **search for** reusable code?
  - Tools that search open-source repos:
    - GitHub, Krugle, Open Hub, Merobase, OpenGrok, ...

Implementation: Build

- **Option 3: Build**
  - (pro) Complete control
  - (con) Complete responsibility!
- Option 3a: **Compose** new code
  - The focus of academic programming
- Option 3b: **Reuse** existing code



Implementation: Build

Kinds of reuse

- Copy and paste
  - But beware of excessive code cloning
- Call a library function
- Use a library class
  - Create your object to **call** a static method of a library class
  - Composition: Create your object such that it's **composed** of an object of a library class
  - Inheritance: Define your class such that it **inherits** from a library class

Implementation: Build

- The ***Reusability Paradox***
  - From the field of "learning objects"
  - **Large** modules do **more** work, but can be used in **fewer** situations
  - **Small** modules do **less** work, but can be used in **more** situations
- Designing for reuse inherently involves compromise

David Wiley.
"The Reusability Paradox."
http://cnx.org/content/m11898/latest/

47

Implementation: Build

So, reusing code is a good implementation option
The implication is that you should try to design code such that it's easy to reuse
That's difficult because of…

[see slide]

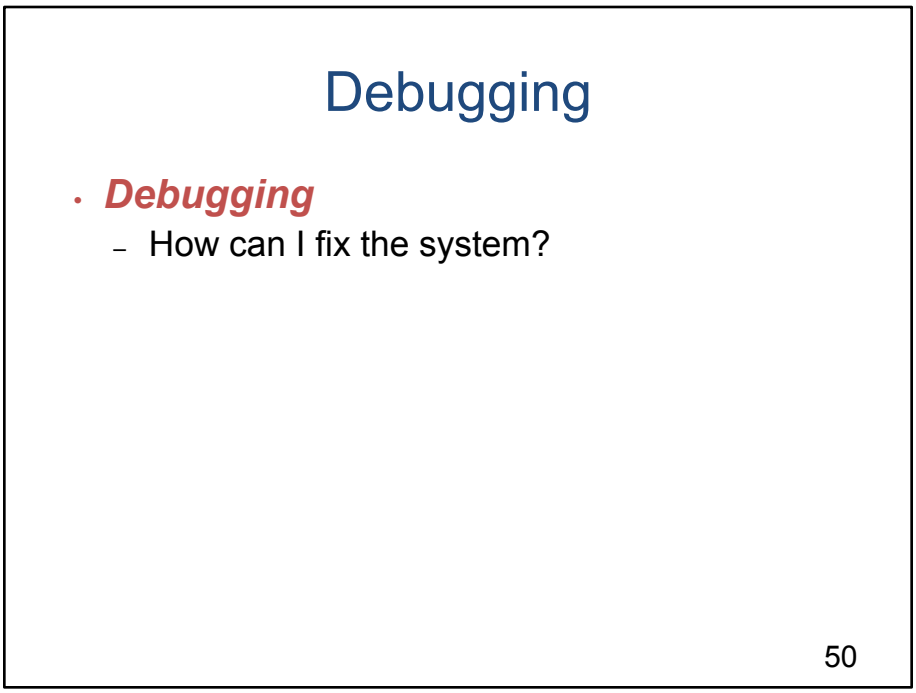
Example: Bluestone course development

The reusability paradox applied to software development too
Designing software for reuse inherently involves compromise
The only way to make reasonable decisions is to know a lot about the users

You've implemented your system in code.
What's next?

Agenda

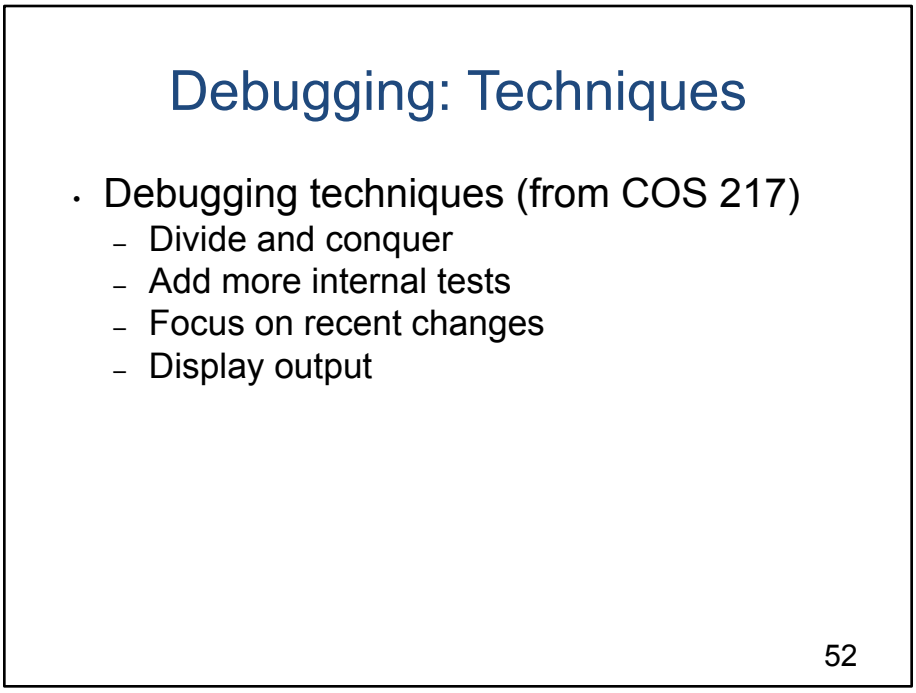
- Requirements analysis
- Design
- Implementation
- **Debugging**
- Testing
- Evaluation
- Maintenance
- Process models



Debugging

[see slide]

Mostly review of debugging techniques covered in COS 217
I'll add a little at the end



Debugging: Techniques

Debugging techniques (from COS 217)

- Divide and conquer
  - Incrementally find smallest input file that illustrates the bug
  - Incrementally find the smallest client code subset that illustrates the bug
  - Use a “binary search” approach
- Add more internal tests
  - Internal tests can help reveal bugs (see Testing)
  - Internal tests can help eliminate bugs
  - Validating parameters & checking invariants can eliminate some functions/methods/modules from the bug hunt
- Focus on recent changes
  - Hard: change code, note new bug, remember what changed
  - Easier: backup code, change code, note new bug, compare new code with old to determine what changed
  - Corollary: Use a version control system
- Display output
  - Remember to write to stderr, not stdout
    - Dangerous to write error message to stdout
    - Stdout is buffered
    - Program could crash before error message is displayed
    - Instead write error message to stderr

Or maybe better still, write to a log file

Assignments:

Server should write log messages to stdout/stderr

Better to write log messages to a log file

Projects:

Same

(Heroku automatically saves messages that your app writes to stdout/stderr)

Debugging: Techniques

- Debugging techniques (from COS 217)
  - Use a debugger

Language	Debugger	Reference
C	**gdb**	COS 217
Python	**pdb**	Appendix of *The Python Language (Part 5)*
Java	**jdb**	https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html

Debugging: Techniques

[see slide]

I use gdb heavily
I rarely use pdb
I rarely use jdb

Debugging: Techniques

- Debugging techniques (not from COS 217)
  - Use an **issue tracking system**
    - **Examples**: Issues (GitHub), Bugzilla, Jira, Trello, Trac, …
    - See https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems

53

Debugging: Techniques

Debugging techniques (not from COS 217)
Use an issue tracking system
Allows users/pgmmers to report issues
Allows assigning of issues to pgmmers
Tracks issues through their life cycles
Maintain history of each issue
Generates summary reports
Examples: Issues (GitHub), Bugzilla, Jira, Trello, Trac, …
See https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems

You're reasonably sure that your code is bug-free. What's next?

Continued in
Software Engineering (Part 3)…