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

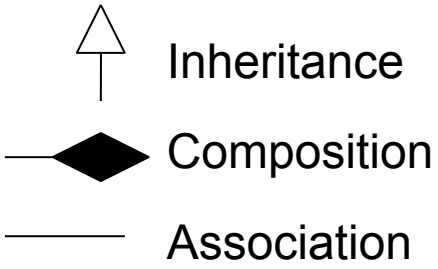
Design: OO Heuristic 1

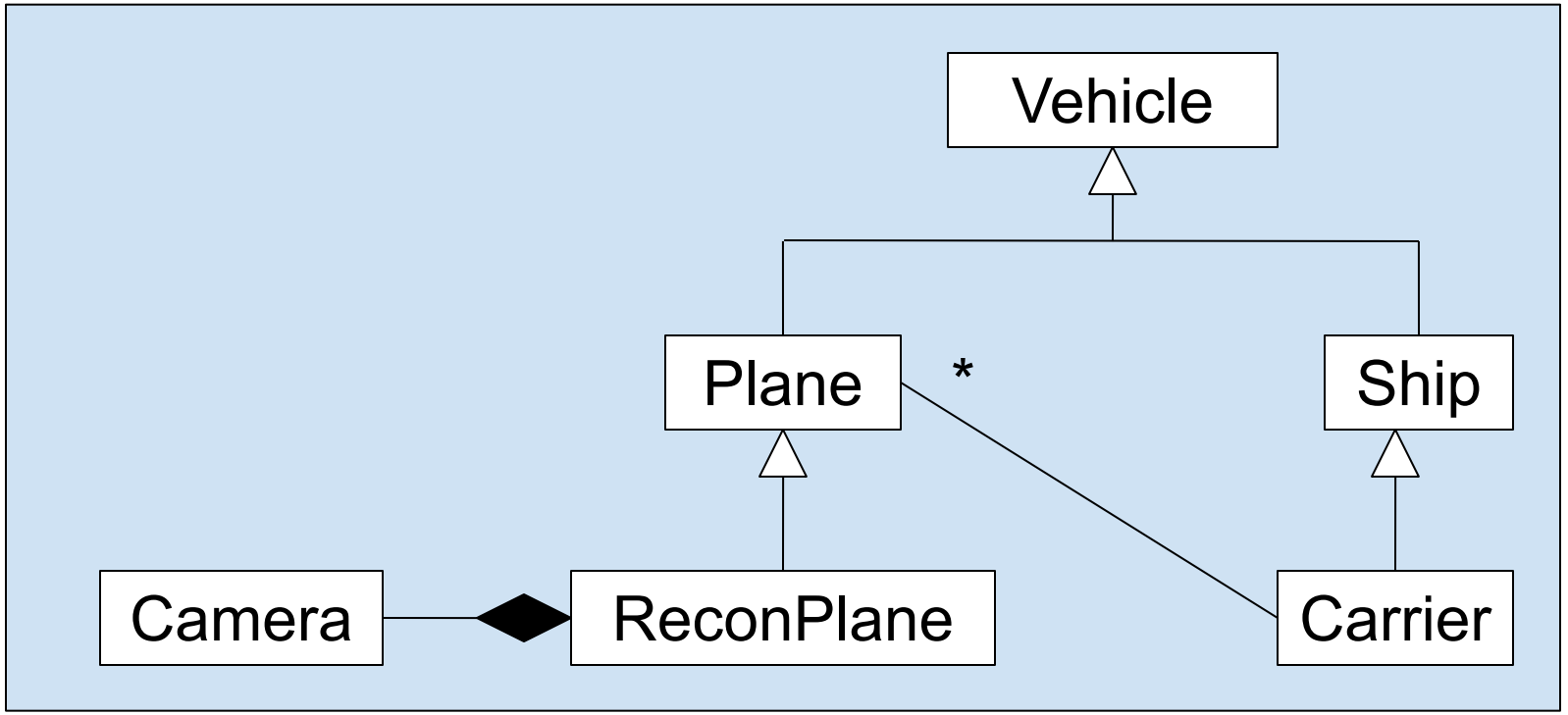
- Use **inheritance** to model "**is a**"
  - Or "**is a kind of**"
- Use **composition** to model "**has a**"

- Examples…

Design: OO Heuristic 1

Recall:

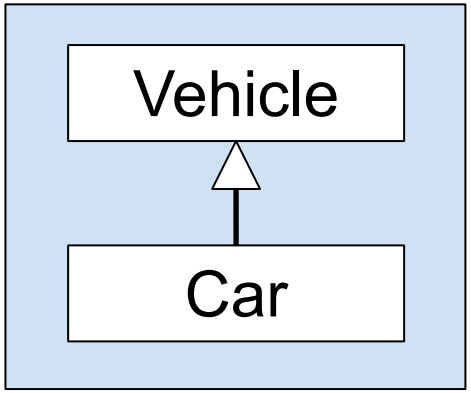




Design: OO Heuristic 1

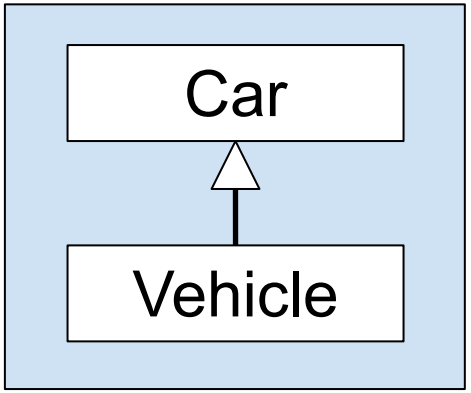
Which is proper?

A



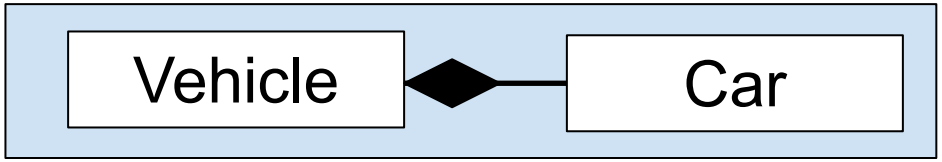
Car inherits from Vehicle

B



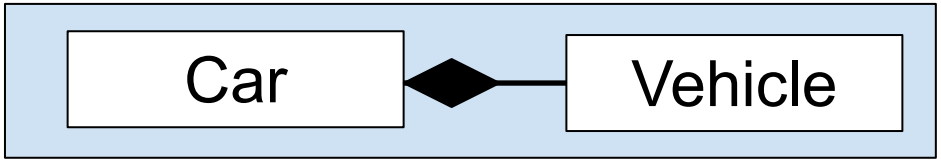
Vehicle inherits from Car

C



A Vehicle object is composed of a Car object

D

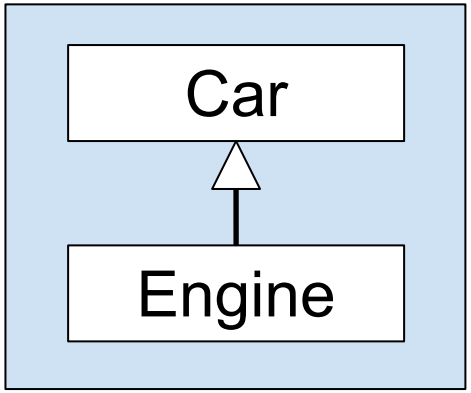


A Car object is composed of a Vehicle object

Design: OO Heuristic 1

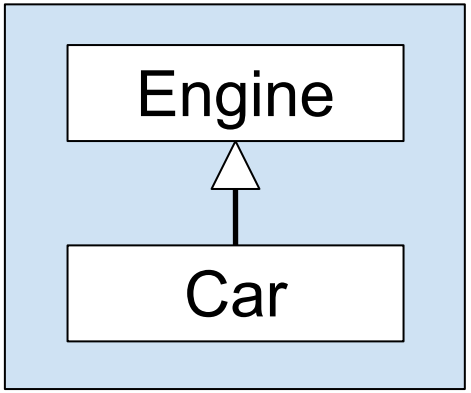
Which is proper?

A



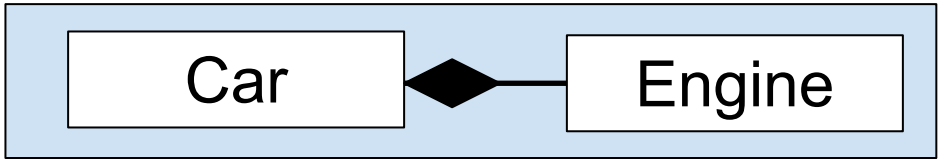
Engine inherits from Car

B



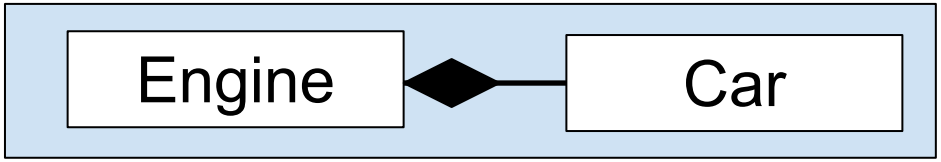
Car inherits from Engine

C



A Car object is composed of an Engine object

D



An Engine object is composed of a Car object

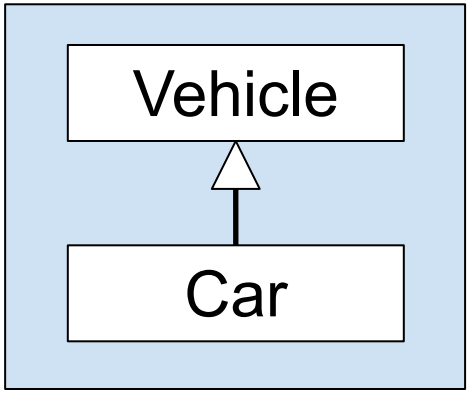
Design: OO Heuristic 2

- 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

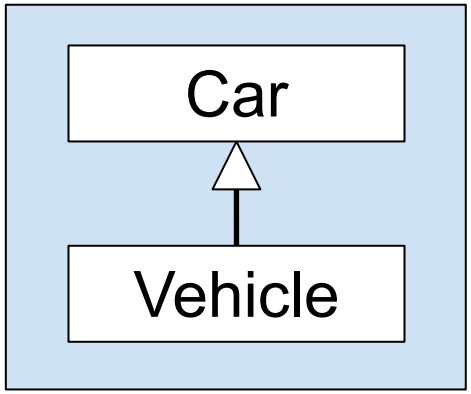
Barbara Liskov and Jeannette Wing.
"A behavioral notion of subtyping,"
ACM Transactions on Programming Languages and Systems,
Volume 16, Issue 6 (November 1994), pp. 1811 - 1841.

Design: OO Heuristic 2

- Use the Liskov sub principle (cont.)



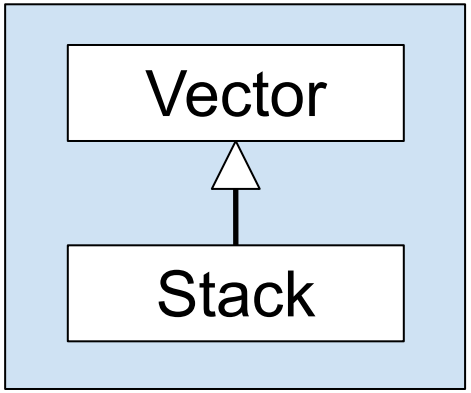
Suppose we have some code that uses a Vehicle object (of some kind)
Can we can replace the Vehicle object with a Car object and expect the code to work?
Yes!



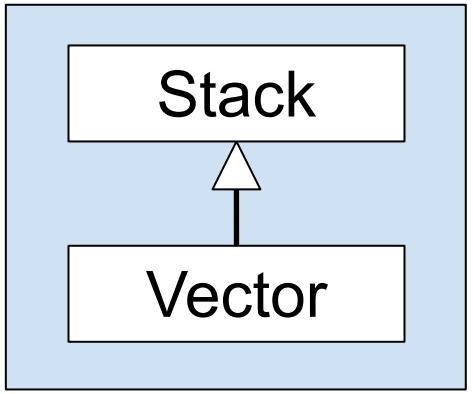
Suppose we have some code that uses a Car object
Can we can replace the Car object with a Vehicle object (of any kind) and expect the code to work? **No!**

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

- Favor composition over inheritance
  - Inheritance
    - ***White box reuse***
  - Composition:
    - ***Black box reuse*** => safer

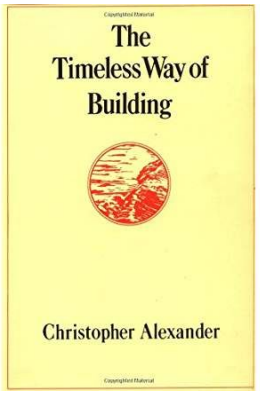
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

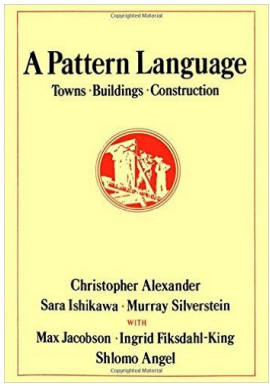
Design: OO Heuristic 4

- Use **OO design patterns**…

Aside: Architectural Patterns

Christopher Alexander





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**

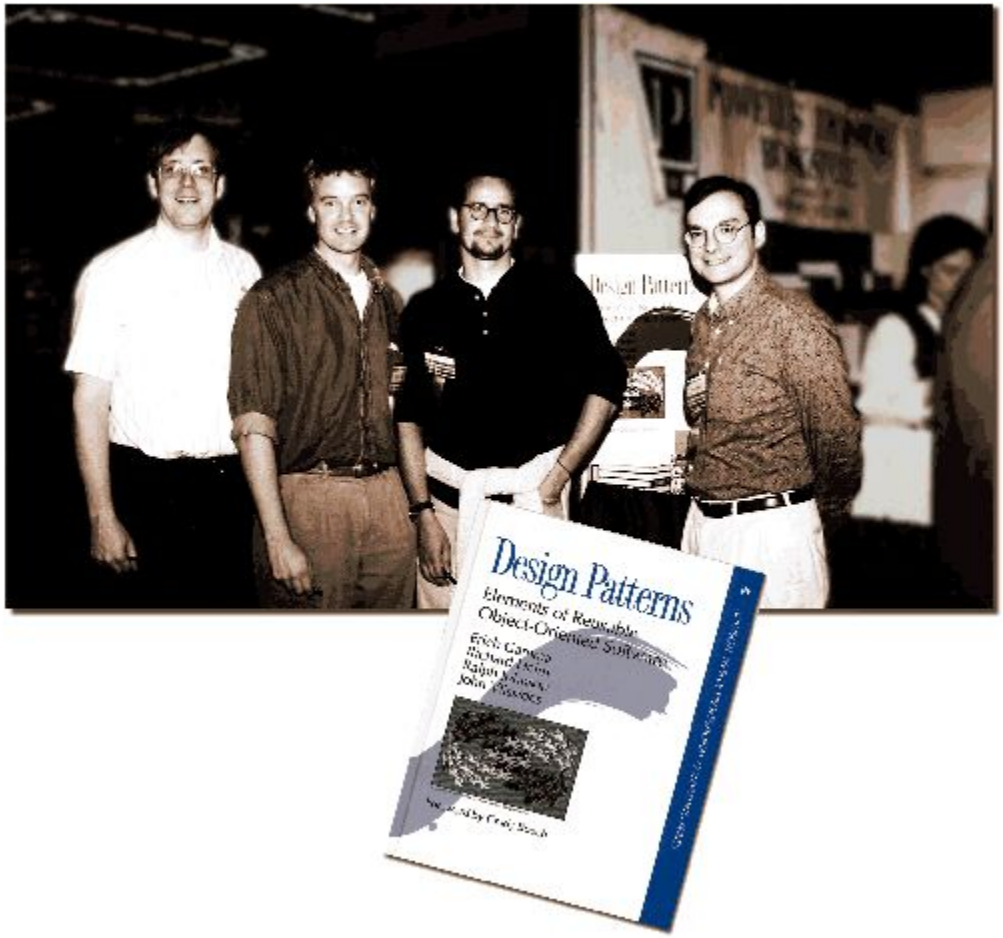
"The relationship of a house to a street is often confused: either the house opens entirely to the street and there is no privacy; or the house turns its back on the street, and communion with street life is lost."

"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.

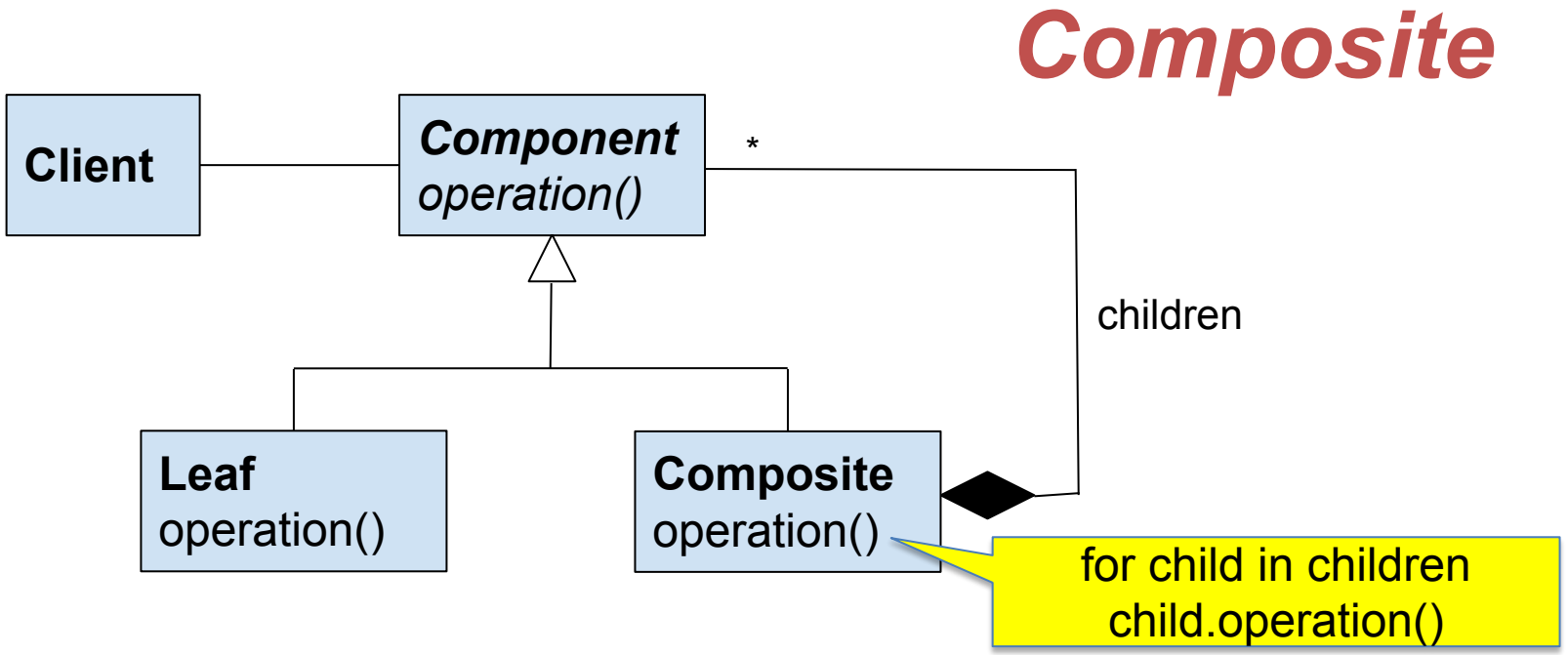
Design: OO Pattern: Composite

- 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."

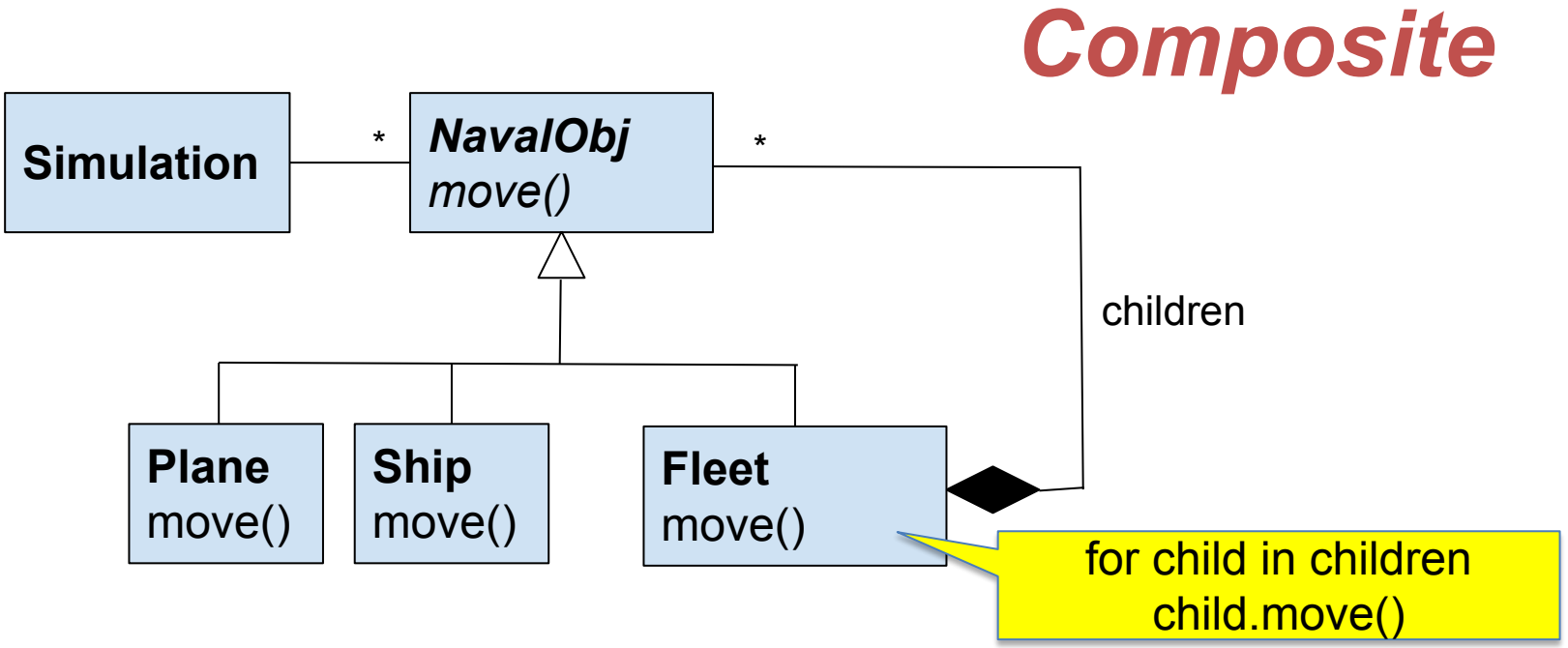
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. Reading, MA. 1995.

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

- 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.

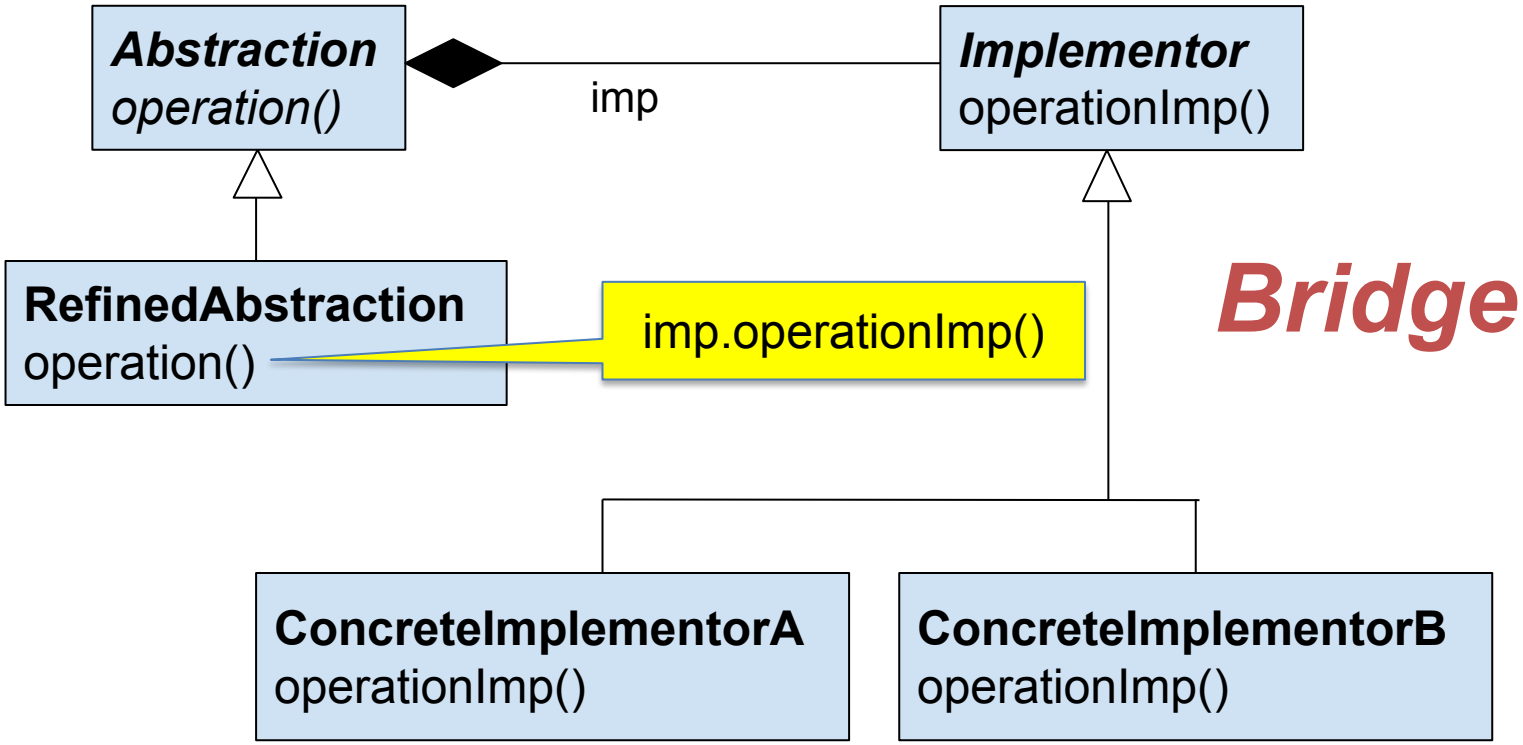
Design: OO Pattern: Bridge

- Example: ***Bridge*** (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 case, 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.

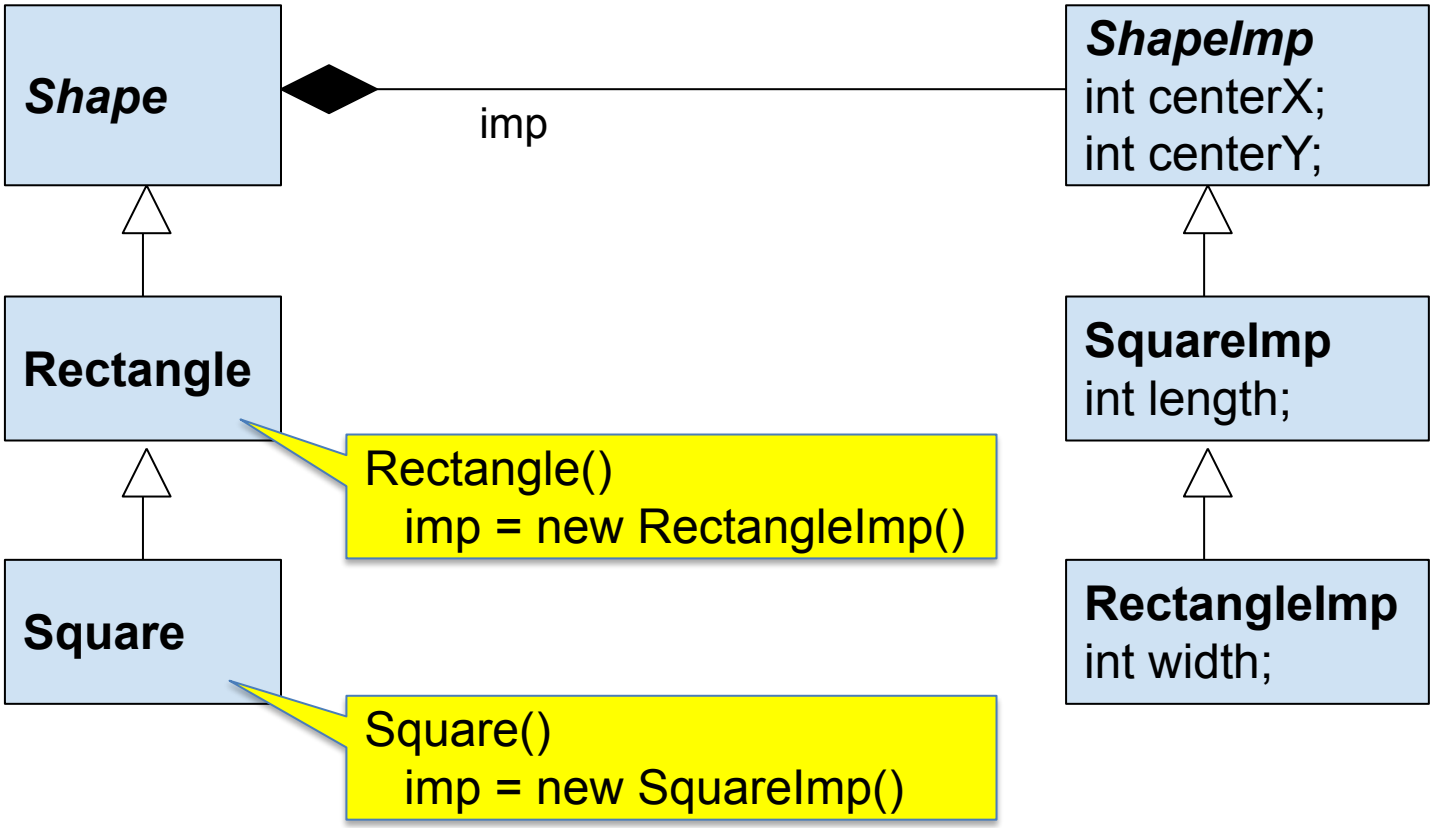
Design: OO Pattern: Bridge



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***
  - Coding the system yourself is only one option

Implementation

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

Implementation

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

Implementation

- **Option 3: Build**
  - (pro) Complete control
  - (con) Complete responsibility!

- Option 3a: **Compose** new code
  - The focus of academic programming
- Option 3b: **Reuse** existing code
  - Use code that you (or someone in your company) previously composed

Implementation

- The ***Reusability Paradox***
  - **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/

If you decide to **build** the system, how should you do it?

Implementation

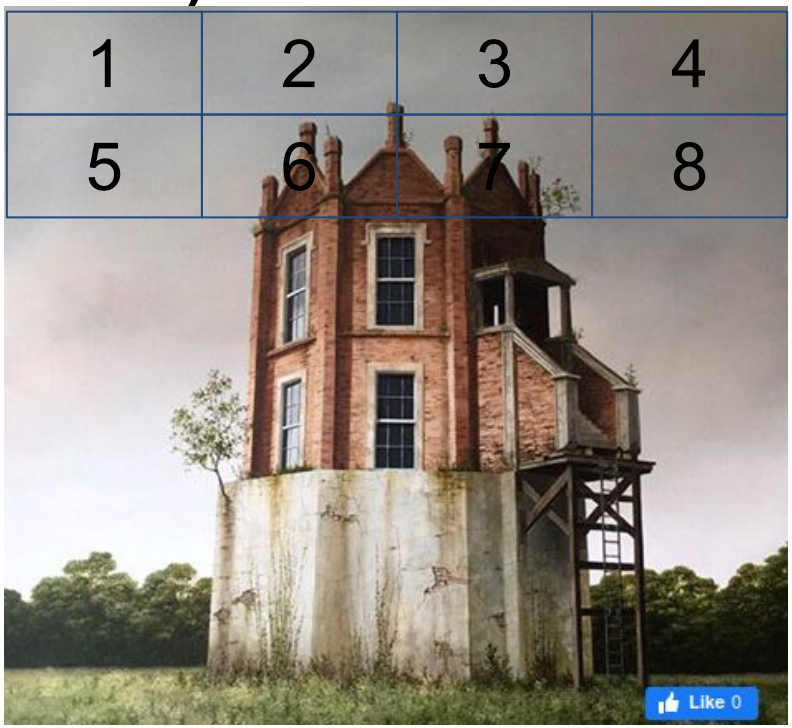
- ***Bottom-up design*** 😕
  - Compose one part of the system in detail
  - Compose another part of the system in detail
  - Repeat until finished

Implementation

- **Bottom-up design** in artistic painting
  - Paint part of painting in complete detail
  - Paint another part of painting in complete detail
  - Repeat until finished

Implementation

- **Bottom-up design** in artistic painting (cont.)



Unlikely to produce a good painting

https://www.demilked.com/sketch-vs-final-product/

Implementation

- **Bottom-up design** in programming
  - Compose part of program in complete detail
  - Compose another part of program in complete detail
  - Repeat until finished

  - Unlikely to produce a good program

Implementation

- ***Top-down design*** 😊
  - Compose entire system with minimal detail
  - ***Successively refine*** until finished

Implementation

- **Top-down design** in artistic painting
  - Sketch the entire painting with minimal detail
  - Successively refine until finished

More likely to produce a good painting

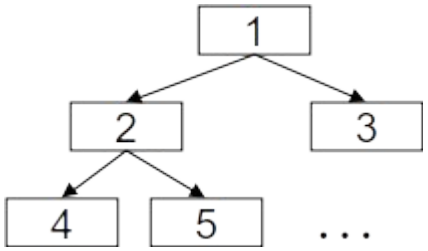
https://www.demilked.com/sketch-vs-final-product/

Implementation

- **Top-down design** in programming
  - Compose main() function in pseudocode with minimal detail
  - Refine each pseudocode statement
    - Small job => replace with code
    - Large job => replace with a function call
  - Repeat until finished
  - Yields good modularity
    - Each function does a small well-defined job

Implementation

- **Top-down design** in programming (cont.)



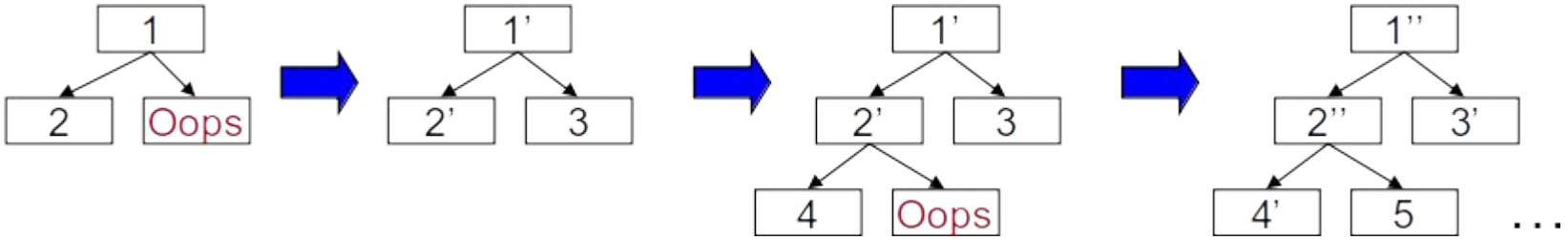
More likely to produce a good program
Bonus: program is naturally modular

Implementation

- **Top-down design** in programming **in reality**
  - Compose main() function in pseudocode
  - Refine each pseudocode statement
    - Oops! Details reveal design error, so…
  - **Backtrack** to refine existing (pseudo)code, and proceed
  - Repeat in (mostly) breadth-first order until finished

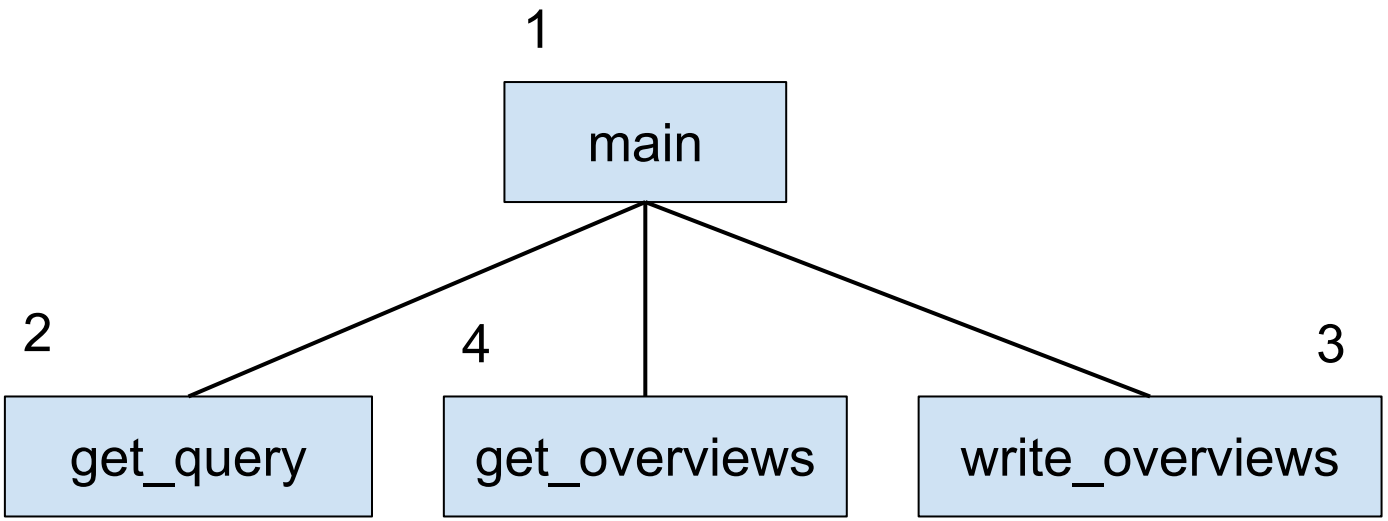
Implementation

- **Top-down design** in programming **in reality** (cont.)



Implementation

- **Top-down design** example
  - Asgt 1: regoverviews.py



Implementation

- **Top-down design** example
  - Asgt 1: regoverviews.py

```
def main():
    try:
        query = get_query()
        classes = database.get_overviews(query)
        write_overviews(classes)
    except Exception as ex:
        print(sys.argv[0] + ': ' + str(ex),
            file=sys.stderr)
        sys.exit(1)
```

Program is modular

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

Agenda

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

Debugging

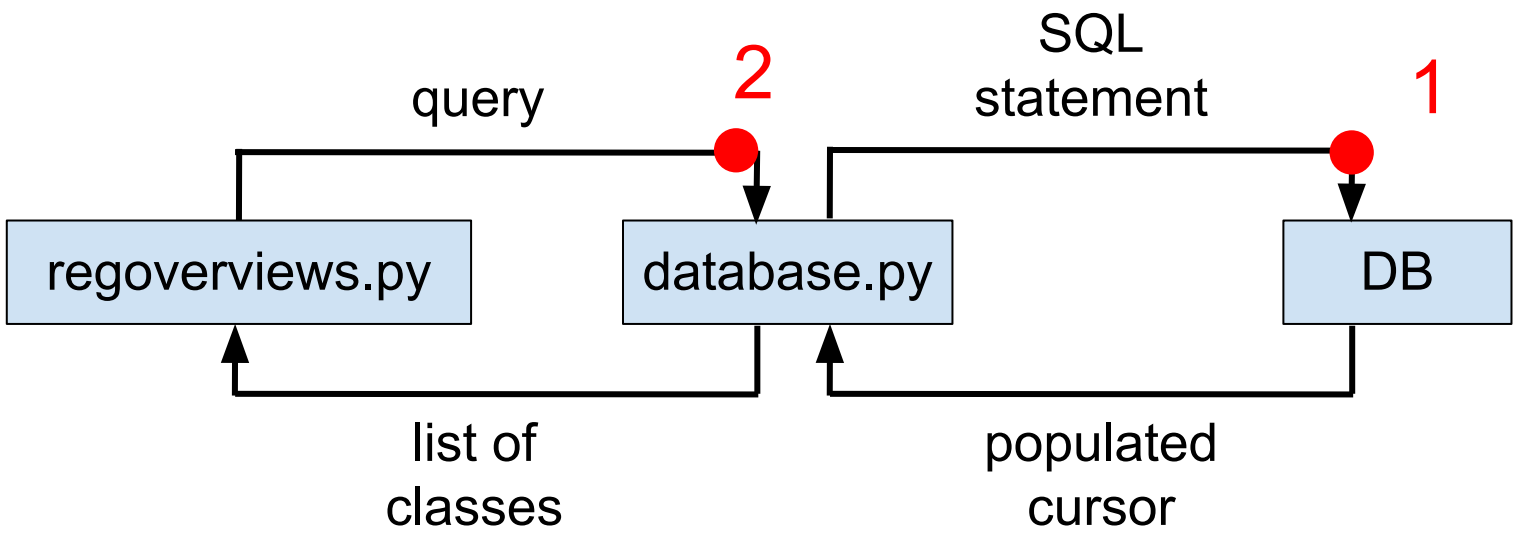
- ***Debugging***
  - How can I fix the system?

Debugging

- Debugging techniques (from COS 217)
  - Divide and conquer

Debugging

Asgt 1:



Debugging

- Debugging techniques (from COS 217)
  - Add more internal tests
  - Focus on recent changes
  - Display output

Debugging

- 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
JavaScript	**Chrome** **Firefox** **…**	https://www.w3schools.com/js/js_debugging.asp
JavaScript	**Node.js**	https://nodejs.org/api/debugger.html

Debugging

- Debugging techniques (not from COS 217)
  - Use an **issue tracking system**
    - **Examples**: Issues (GitHub), Bugzilla (open source), Jira (Atlassian), Trello (Atlassian), Trac (open source), …
    - 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)…