Software Design

(patterns, heuristics)

Dennis Mancl

dmancl@acm.org

MSWX

MSWX ◊ Mancl ◊ Software ◊ Experts ◊ http://manclswx.com

Design is hard

How to make better designs?

- We will consider some design heuristics first…
  - A set of simple guidelines for a complex task
    - We create heuristics to help us think…
    - Heuristics help us to make choices
  - Useful for software design
    - Like “coding rules” – they can help us get started
    - When we have two “designs” – it isn’t easy to know which one is the best

Heuristics = simple guidelines

Focus on good design

Elements of flexible designs

- Design principles: good but complicated…
  - **Modularization** – divide a big design into small independent pieces (component / module)
  - **Encapsulation** – each component has a simple interface, hiding complex details (reduce the learning curve…)
  - **Low coupling** – minimize dependencies between components
  - **Extensibility** – components can be extended without changing the base code (making it easier to add new operations later…)

Modules

Modules help you understand your own design later

Modules help with teamwork

- Why are modules important?
  - Modularization is a great design idea!
  - “Divide and conquer”
  - Modules =
    - Packages, Components, Classes
- Design heuristics help us define “good” modules
  - In object oriented design, we use “classes” to build modules

Classes are bundles of data and related operations

Good modular classes! (Not just functions in disguise)

Arthur Riel's Object Oriented Design Heuristics

- Some good design heuristics:

Object-Oriented Design Heuristics
by Arthur Riel

- This book lists 61 heuristics
  - these are "rules of thumb"
  - they are sometimes violated in designs that are considered good
  - but the heuristics help identify places where the design should be changed
- These heuristics apply to many programming languages
  - C++, Java, Python, JavaScript…

Some important “basic” heuristics

- Arthur Riel’s heuristics cover basic rules for modules:
  - Make all class data “private”
    - H2.1: All data should be hidden within its class.
  - Maintain a simple interface to a reusable class/module
    - H2.2: Users of a class must be dependent on its public interface, but a class should not be dependent on its users.
  - Don’t make a class too complicated
    - H2.6: Don’t clutter the public interface of a class with things that the user of a class can’t use, shouldn’t use, or probably won’t be interested in using.
  - Don’t mix multiple abstractions in a single class
    - H2.8: A class should capture one and only one key abstraction.

Encapsulation

- H2.1: All data should be hidden within its class.

```
class Point { /* C++ example */
private:
   int x_coordinate;
   int y_coordinate;
public:
   void setposition(int x, int y) {
      x_coordinate = x;  y_coordinate = y;
   }
   void moveposition(int delta_x, int delta_y) {
      x_coordinate += delta_x;  y_coordinate += delta_y;
   }
   int getx() const {
      return (x_coordinate);
   }
   int gety() const {
      return (y_coordinate);
   }
};
```

Data is "hidden" as private data attributes – we don't want the data to be public

In the public section of the class, we have "accessor" and "modifier" functions to be used by others

Encapsulation == we must use the public interface to read and write data

Encapsulation

Encapsulation in Java!!

- H2.1: All data should be hidden within its class.

```
public class Point { /* Java example */
  private int x_coordinate;
  private int y_coordinate;

  public void setposition(int x, int y) {
    x_coordinate = x;  y_coordinate = y;
  }
  public void moveposition(int delta_x, int delta_y) {
    x_coordinate += delta_x;  y_coordinate += delta_y;
  }
  public int getx(){
    return (x_coordinate);
  }
  public int gety(){
    return (y_coordinate);
  }
}
```

Data is "hidden" as private data attributes – we don't want the data to be public

In the public section of the class, we have "accessor" and "modifier" functions to be used by others

Encapsulation == we must use the public interface to read and write data

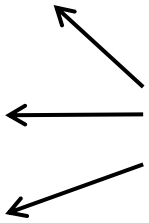
Encapsulation

Encapsulation in Python!!

- H2.1: All data should be hidden within its class.

```
class Point:  # Python example
  def _ _init_ _(self):
    self._ _x_coordinate = 0
    self._ _y_coordinate = 0

  def setposition(self, x, y):
    self._ _x_coordinate = x;  self._ _y_coordinate = y

  def moveposition(self, delta_x, delta_y):
    self._ _x_coordinate += delta_x
    self._ _y_coordinate += delta_y

  def getx(self):
    return self._ _x_coordinate

  def gety(self):
    return self._ _y_coordinate
```

Data is "hidden" as private data attributes – we don't want the data to be public



We have public "accessor" and "modifier" functions to be used by others

Python naming convention for data hiding: attributes beginning with two underscores are "private"

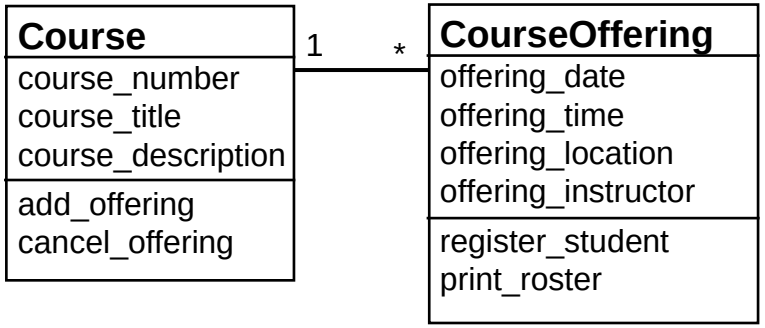
Encapsulation == we must use the public interface to read and write data

Keep the public interface clean

- H2.6: Don't clutter the public interface of a class with things that the user of a class can't use, shouldn't use, or probably won't be interested in using.

Maintain a sorted list of all offerings for this course

```
class Course {
private:
    string course_number;
    string course_title;
    string course_description;
    vector<CourseOffering*> course_offerings;
public:
    Course(string cnum, string ctitle, string cdesc);
    void add_offering(CourseOffering *coffering);
private:
    Course *find_position_to_insert(CourseOffering *coffering);
};
void Course::add_offering(CourseOffering *c)  {
    Course *insert_pos =
        find_position_to_insert(c);
    if (position_to_insert != 0) {
        course_offerings.insert(insert_pos, c);
    }
}
```



This private function is a helper function – it will search for the position to insert the new CourseOffering object into the list.

It would be a mistake to make it a "public" function, because external users of the Course class should never need to call it…

- "ordering" of the list is an Internal Secret

Look out for “god classes”

- A “god class” is an attempt to build a design with centralized control
  - there is a single main procedure that is “in command” of the control flow
  - with application’s data *shared* between many different procedures
- Action-oriented programs don’t always evolve gracefully...
  - watch for “accidental complexity”
  - when we add new functionality to an already-designed system, we often create accidental complexity

Action-oriented => maintenance problems

Action oriented program

An old-fashioned top-down design

- Main program controls everything

```
/* Main program – it gives orders to all of the functions */
int main(int argc, char **argv) {

    initialize_data_structures();
    open_main_database();

    connect_to_external_system_1();
    connect_to_external_system_2();
    connect_to_external_system_3();

    verify_connections();

    while (forever) {
        display_user_interface_screen();
        req = receive_user_request();
        update_database(req);
        send_message_to_external_system(req);
    }
};
```

If any new internal structures are needed, or if we need to connect to a new external system, we must update the main program.

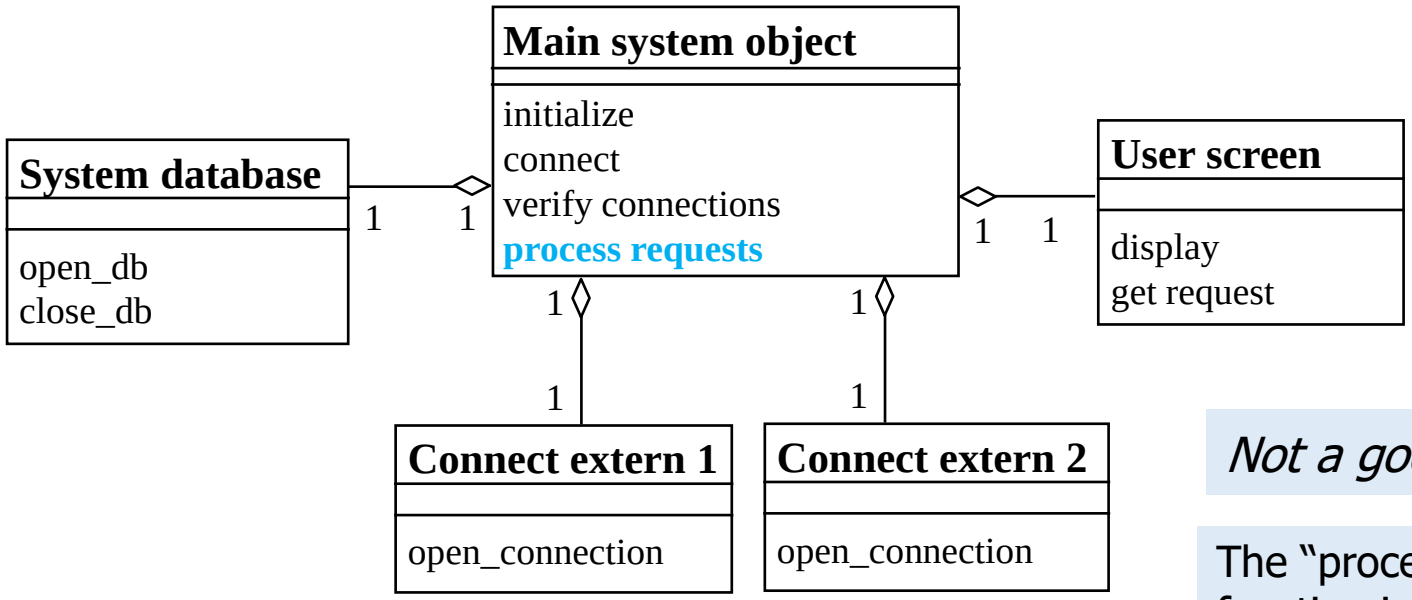
Every critical event must pass through the main program.

Lower-level functions may update global data structures and the database – with no constraints.

Big maintenance headaches…

Object oriented software

- Object oriented software design says: "define classes"
  - easy! … We will make a class for each major function!! (grrr…)
  - ***but***: just because a design is "object-oriented" (the design is composed completely of classes) doesn't mean that it is automatically "good"
  - an application might be superficially object-oriented, while still having the structural problems of an action-oriented application



Not a good design!

The "process requests" function is still huge!

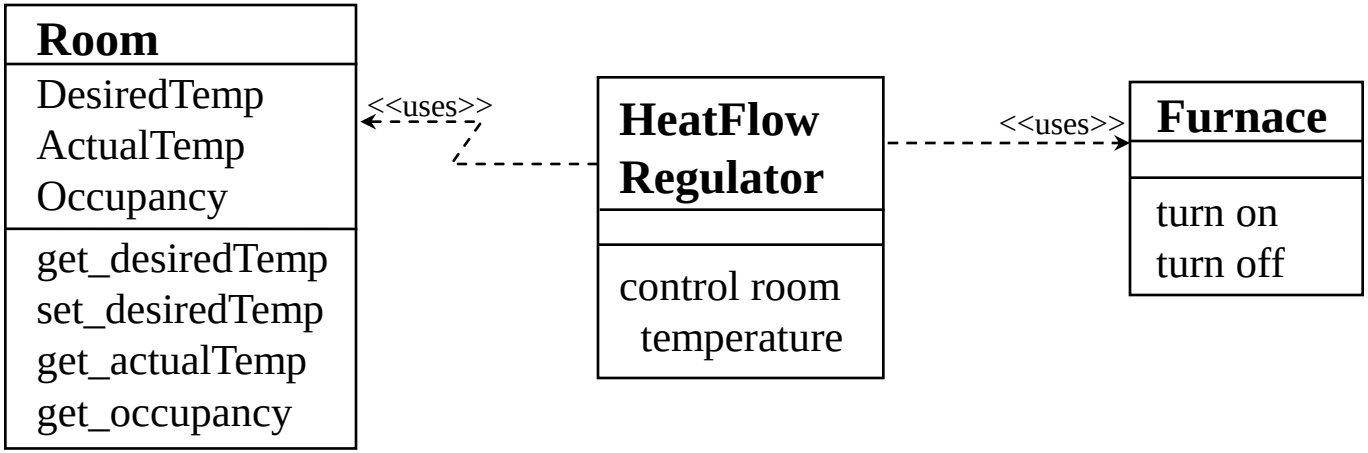
Central control versus object design

- God classes are a sign of poor design
  - The designer is still thinking in the action-oriented paradigm
  - But the designer is trying to recast the design in object-oriented terminology (without really making the transition to an object oriented architecture)

Central control:
code updates can be complex

Example (continued)

- Home heating system (with god class):



Centralized decision-making:

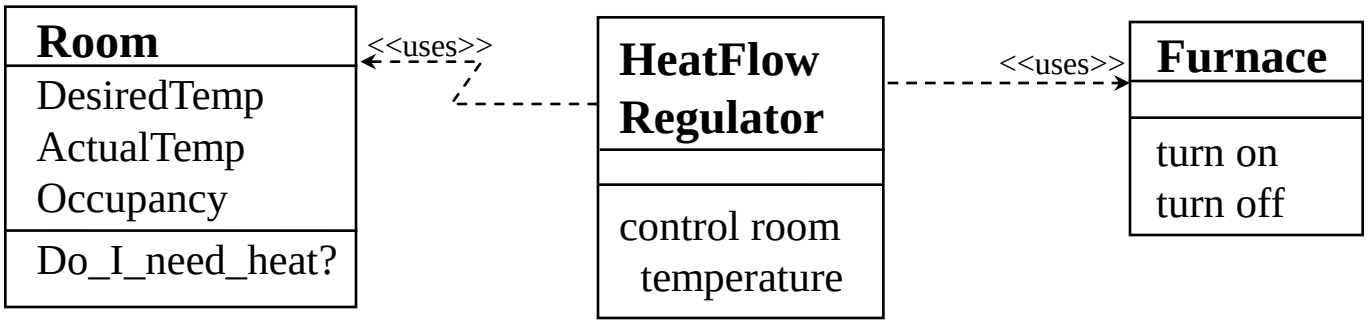
- HeatFlowRegulator accesses data in the Room objects
- Makes the decision to turn on the heat
- Then calls on the Furnace object

Questions about the example

- What's wrong with this design?
  - **HeatFlowRegulator** is an "omnipotent controller" that pulls in all of the information needed to make a decision, and then calls all of the operations that affect the physical objects.
- Is there a better way?
  - Eliminate the "god class" by making the **Room** smarter.

Example (continued)

- Home heating system (without god class):



Decentralized decision-making:

- The Room class is smarter: some of the computation work has been moved to Room
- HeatFlowRegulator still runs the interface to the Furnace

Distribution of functionality

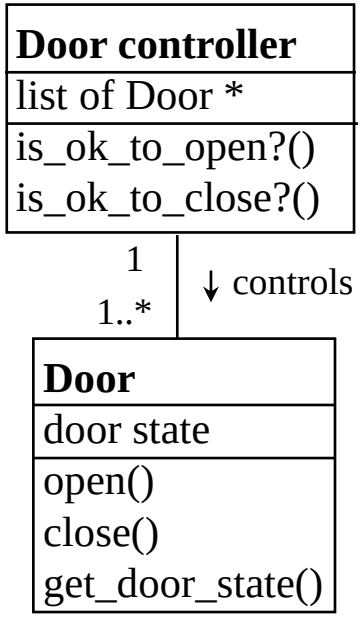
- There are many different kinds of classes in a big system
- And, of course, a system may be designed to use a collection of dumb lower-level classes
  - classes that provide specialized services such as hardware interfaces, formatting of data, database access, and so on
- But in most good object oriented systems, the designers can point to several classes in the design that are “peers”
  - and there ought to be a good distribution of intelligence among these classes
  - *not* just one big god class

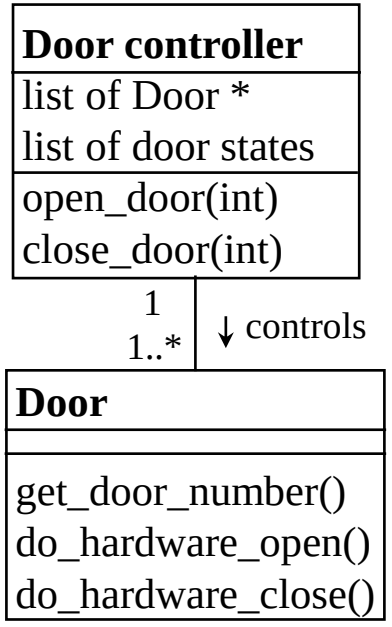
Is the Room class a fundamental part of the home heating system design? Or should it be a “dumb lower-level class”? What do you think?

Example: Design of a Door system

- A simple design example – using design heuristics to evaluate two design alternatives
- In this system, there will be a set of Doors that are being controlled by a Door controller object. The Doors in this system might on…
  - a subway train, a supermarket, a secure building, a space station airlock

Design 1:
Door controller is only responsible for policy

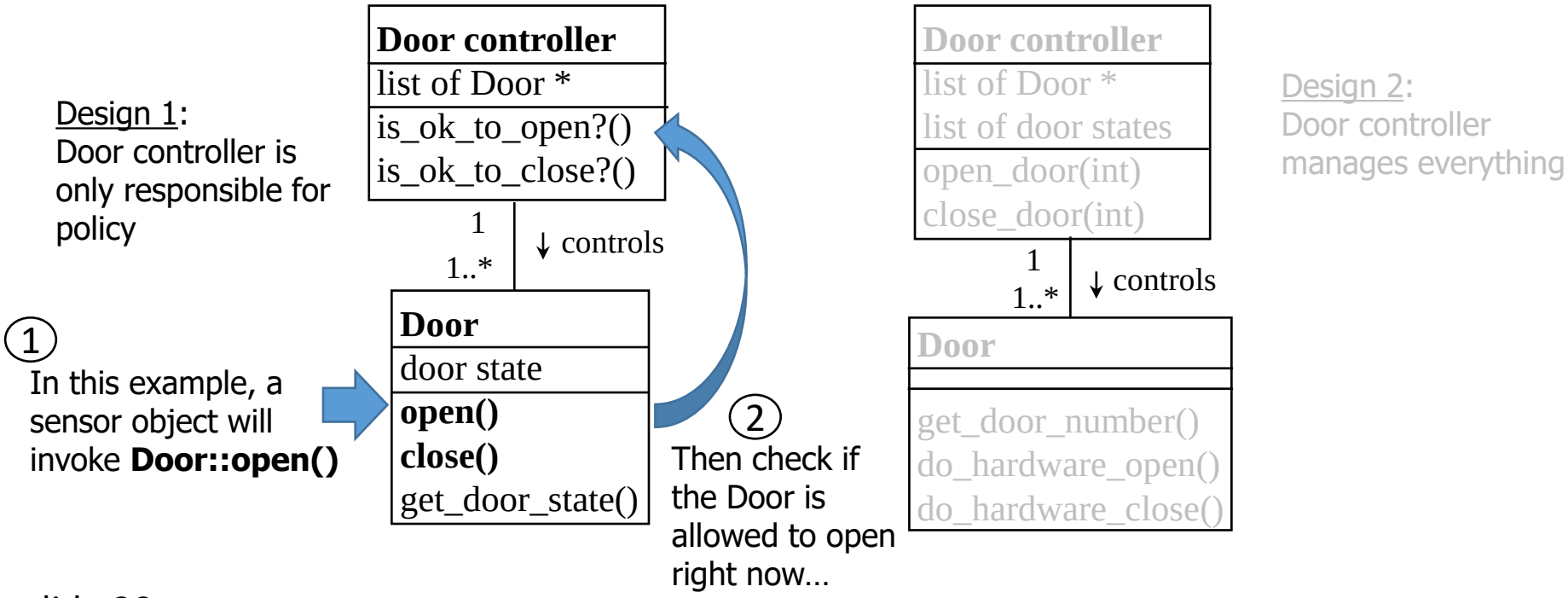




Design 2:
Door controller manages everything

Example: Design of a Door system

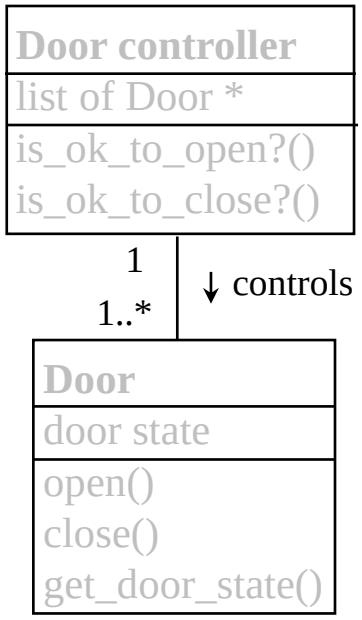
- In Design 1, the Door class provides the public interface.
  - User presses a button on the Door; the Door will ask the Door controller if it is OK to open; if it is OK the open() function will complete successfully
  - The Door controller needs to check the "rules" – can't open a train door unless the train is stopped and in a station…
  - Can't open the door of a secure building unless you have the access code or your ID is in the database



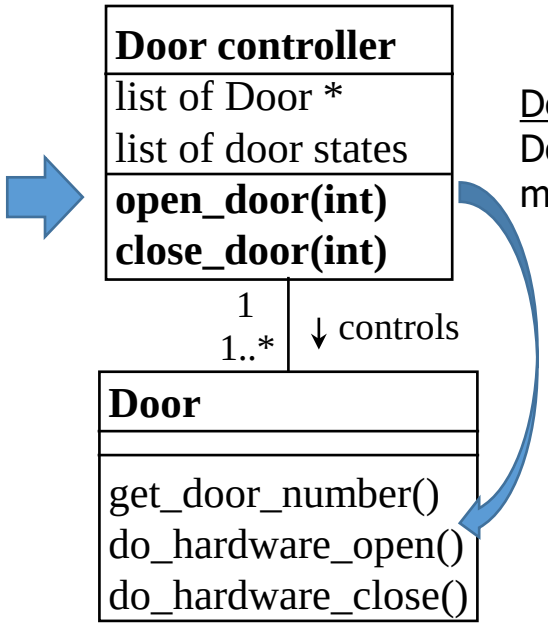
Example: Design of a Door system

- In Design 2, the Door controller class provides the public interface.
  - User sends a command directly to the Door controller; the Door controller checks its rules internally; if everything is OK, it will tell the Door to execute its do_hardware_open()
  - We can say that the Door controller is "directly controlling" each of the Doors

Design 1:
Door controller is only responsible for policy



① In this example, a sensor object will invoke **Door controller::open_door() – Door controller** checks the rules



Design 2:
Door controller manages everything

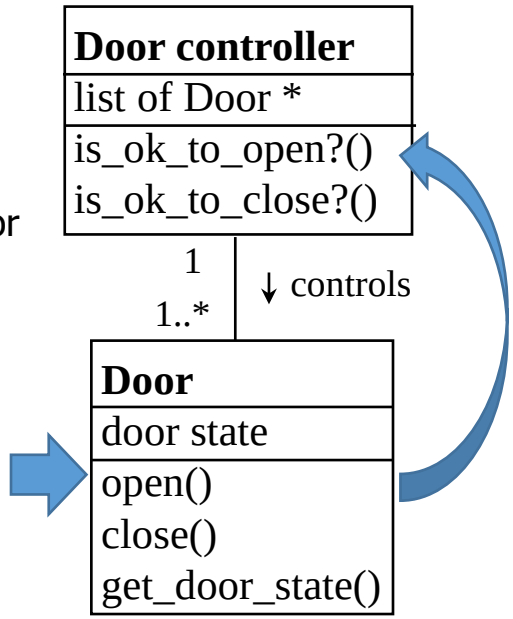
② **Door controller** delegates the low-level operation to the **Door** class

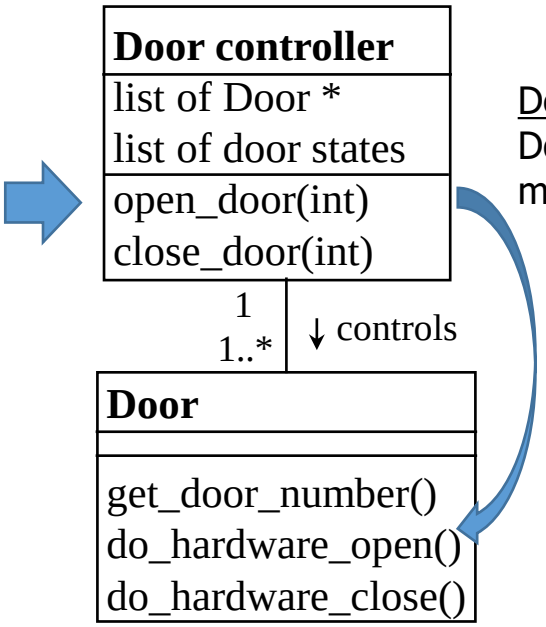
Example: Design of a Door system

Use heuristics to explain what you like or don't like

- Question: Which of the two designs do you prefer?
  - It's your opinion – there is no "right" answer
  - How would you explain your preference in a design review meeting?

Design 1:
Door controller is only responsible for policy





Design 2:
Door controller manages everything

Some possible arguments for each design

- Why is Design 1 better?
  - In Design 2, the Door controller is a “god class”
  - If we make changes to the Door class interface, Design 1 might be better
    - We could have many different models of Door, with different ways for users to make requests: a button on the Door, a motion detector to sense the user moving towards the Door, a wireless interface to allow users to use their cell phones to request opening or closing a Door
    - Each variation could be a “subclass” of the Door class – the interaction with the Door controller is unchanged
- Why is Design 2 better?
  - In Design 2, the scenarios for opening and closing a Door are shorter (and maybe faster)
    - If performance or security is a concern, maybe it is OK to have a god class…
    - Some parts of the Door controller functionality might be built directly in hardware

Design 1 = more flexible; easier to extend

Design 2 = more secure; better performance

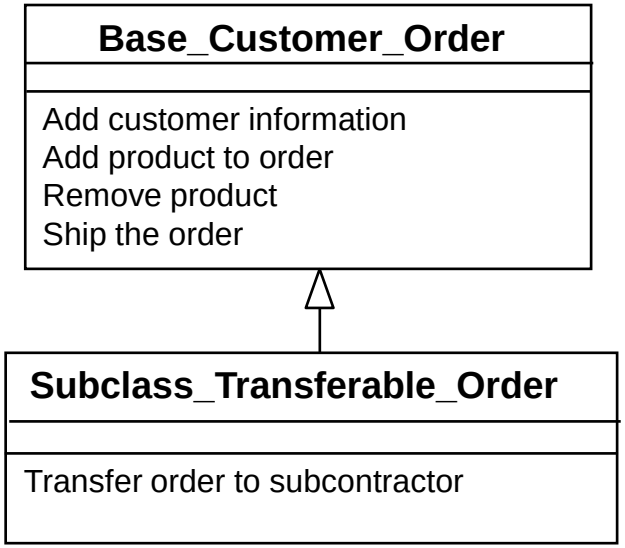
Heuristics are a good way to discuss design alternatives

- We had two possible design alternatives for the Door controller
- In a design review, the participants need to talk about tradeoffs
  - When we see that our design has a god class, we might decide to change it – to keep the design flexible
  - On the other hand, if flexibility is less important than performance, we might choose the god class design – even though it violates one of the design heuristics
- Design heuristics are not absolute rules: they are guidelines that help us think about design alternatives
  - The heuristics are sometimes violated in designs that are considered good

Heuristics help us "critique" a design

Inheritance

- Inheritance (and “subclasses”) is a useful way to define a group of related classes



How are these two classes related?

- Base_Customer_Order is a “base class”
- Subclass_Transferable_Order has *all* of the operations of the base class
- *plus* you can add new operations
- *and* you can redefine the behavior of any of the base class functions

We’re going to talk about “patterns” next – but a lot of Object Oriented patterns use inheritance… so we’d better mention it now! Right away, we will see how it works in the Strategy Pattern…

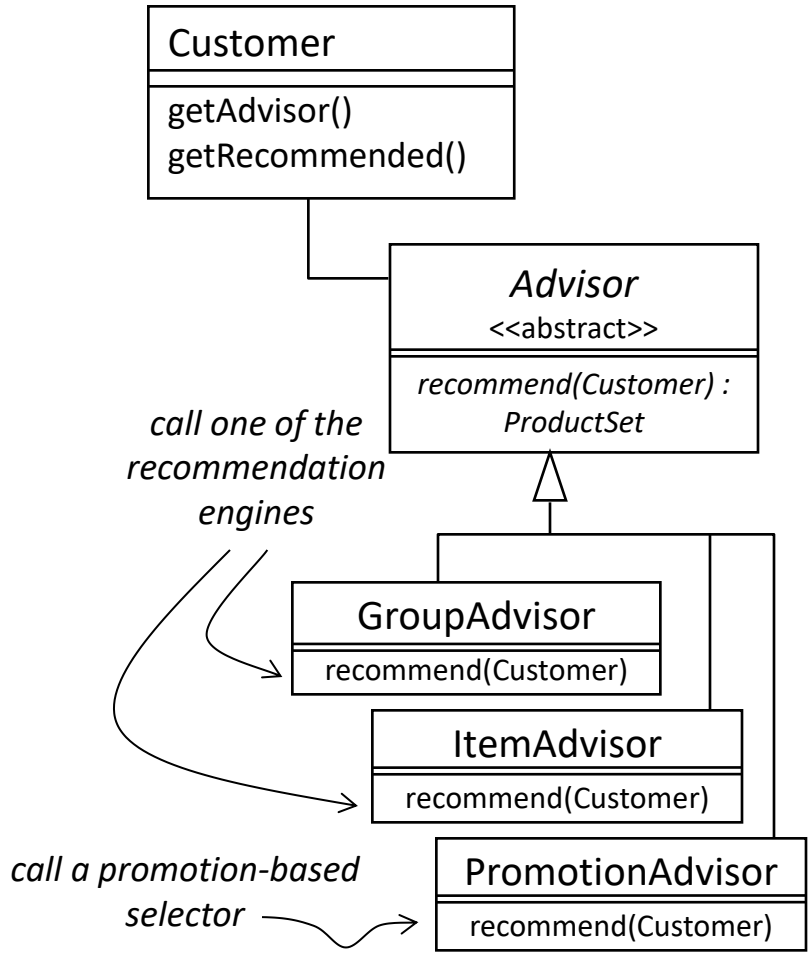
Design Patterns = reusable design ideas

- Design Patterns are “commonly used design elements”
  - design ideas that occur in many different design
- Simple design ideas… but flexible
  - In object oriented design patterns, a Pattern will describe a new group of “classes” to be added to the design
  - Some OO design patterns use “inheritance” (more details later)
  - We’ll start with the Strategy pattern (very simple) and Facade pattern (also very simple)
  - We’ll see the Observer pattern later (a bit more complicated, but it is used a lot!)

Strategy pattern

- **<u>Problem</u>: An application needs to use a family of similar algorithms**
  - the selection of which algorithm depends on the client making the request or some characteristics in the data
- **<u>Solution</u>: Define a family of algorithms, encapsulate each one, and make them interchangeable**
  - there will be a family of classes, one per algorithm variation

Example: Each subclass contains **<u>a different implementation</u>** of the "abstract" recommend function – the function uses information in the Customer class to build a set of recommendations

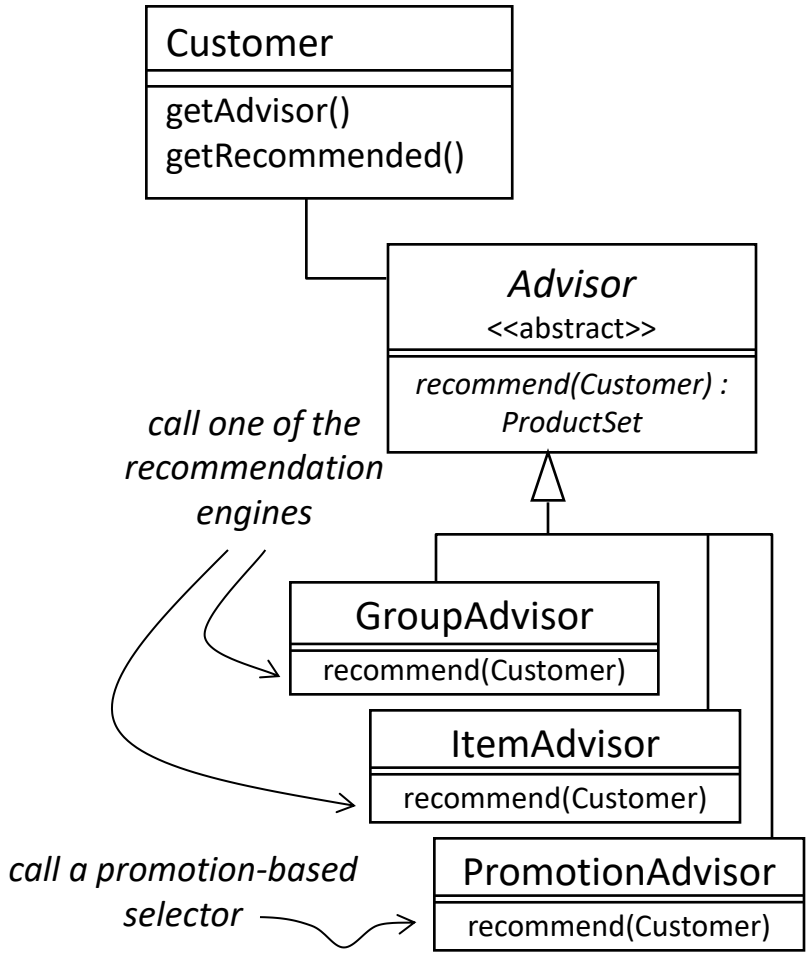


- ***GroupAdvisor:*** *Use data from an "interest survey"*
- ***ItemAdvisor:*** *Use data from customer's previous purchases*
- ***PromotionAdvisor:*** *Seasonal choices*

Strategy pattern

- Think about how this works…
- A customer connects to the website
- There is a “Factory Class” or a “Factory function” that creates a subclass object – depending on the information recorded for that customer
- Throughout the web session, the right kind of “Advisor” will be called to display certain products

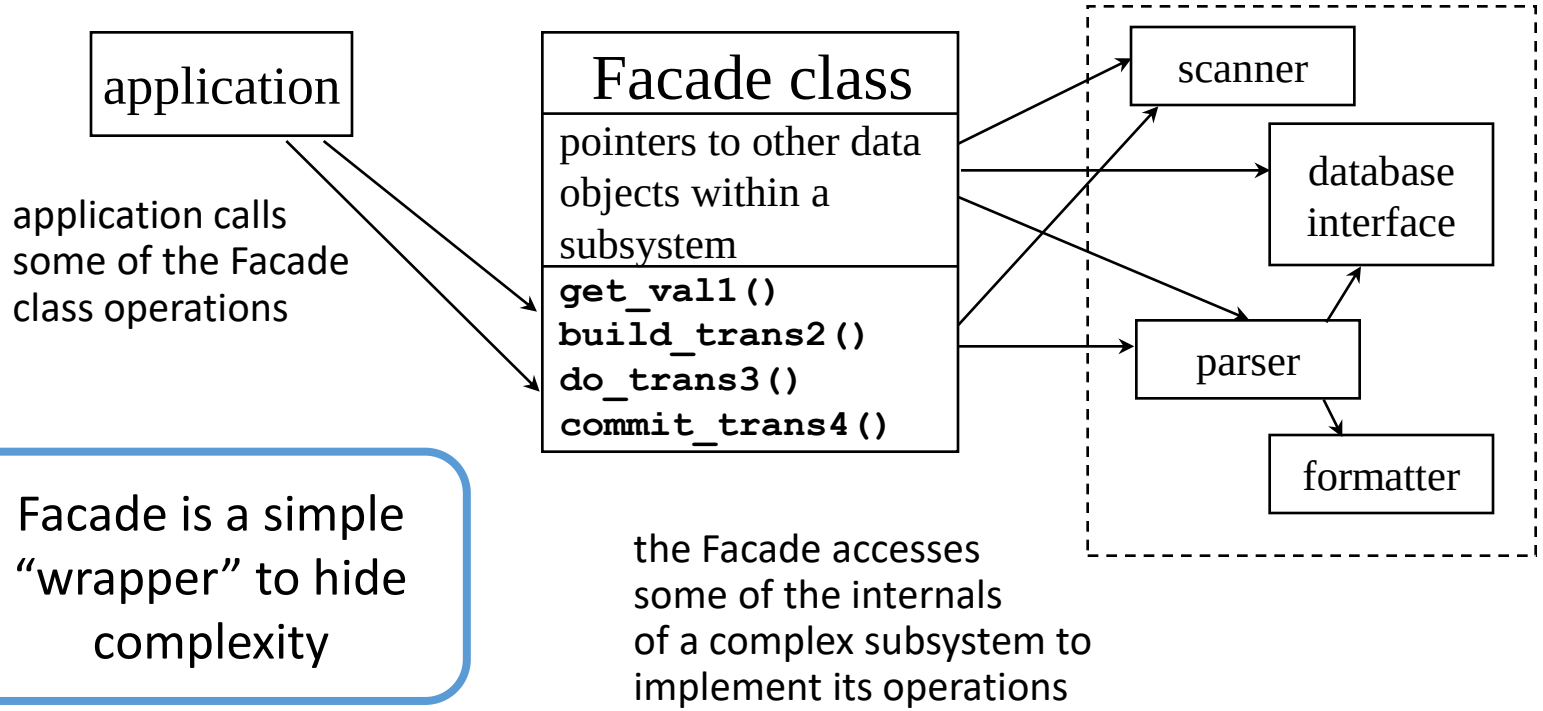
It’s easy to add a new recommendation engine!



- ***GroupAdvisor:*** *Use data from an "interest survey"*
- ***ItemAdvisor:*** *Use data from customer’s previous purchases*
- ***PromotionAdvisor:*** *Seasonal choices*

Facade pattern

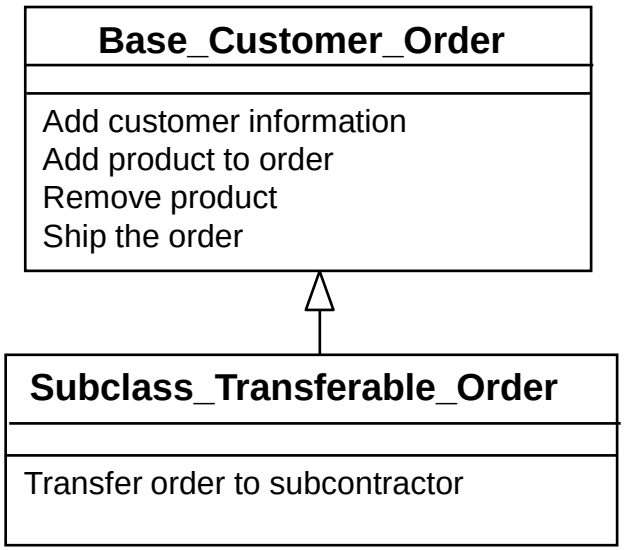
- **Problem**: The application needs a **simple interface** to a complex subsystem
- **Solution**: Create a Facade class provides a simple to use interface – the application developers only need to understand the interface, not the internal details



Facade is a simple "wrapper" to hide complexity

Inheritance – four ways

- Inheritance (and “subclasses”) is a useful way to define a group of related classes



Inheritance = a complex idea in Object Oriented Design

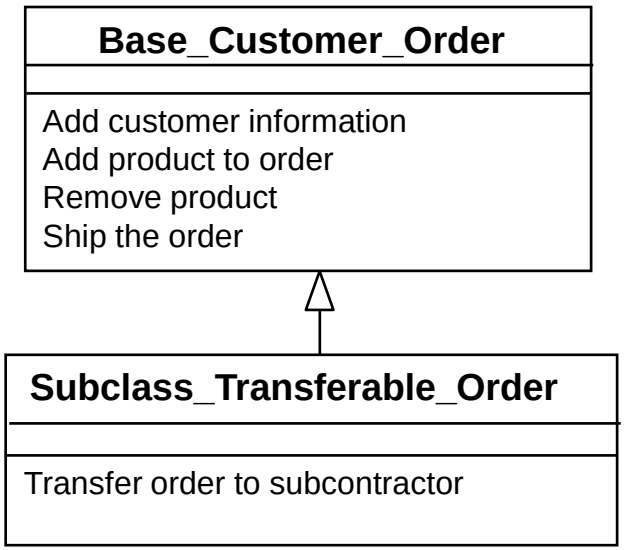
A top-level class defines some functionality.

Subclasses extend or modify that functionality.

- When processing a normal customer order, we add information into a **Base_Customer_Order** record an element at a time, then we “ship” the order.
- Let’s make a fancier “transferable order” that can be reassigned at any time to another processing center…

Inheritance – four ways

- Inheritance (and “subclasses”) is a useful way to define a group of related classes



A subclass “extends” the base class… maybe some “new” behavior.
A subclass can add new operations and it can redefine base class functions

How do we figure out where inheritance is needed in the design?

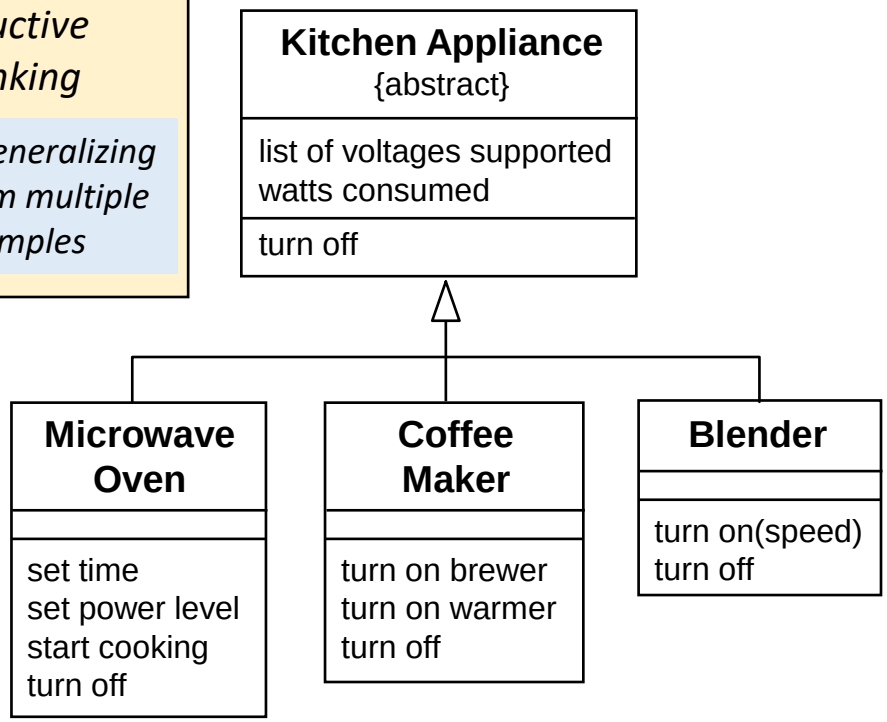
1. Many standard **Design Patterns** use inheritance as a mechanism to invoke “different behavior in different cases”
2. Many **software frameworks** require developers to build application-specific subclasses
3. Do **Commonality analysis**: Abstract interfaces can be defined to support the common behavior in multiple related classes
4. Inheritance to **extend the functionality** of a concrete class

Inheritance (commonality analysis)

- To define…. **Families of similar classes** with some attributes and operations in common (often found in the initial analysis):

Inductive thinking

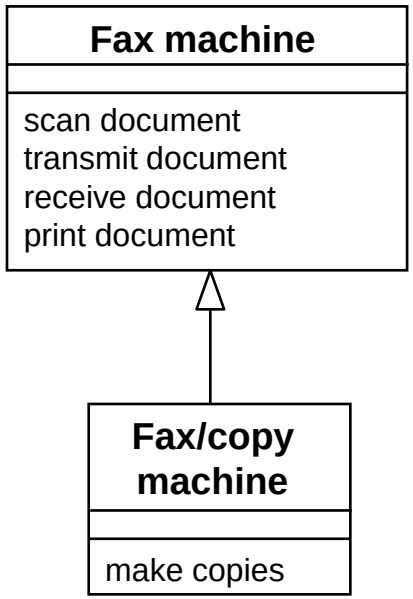
= generalizing from multiple examples



Note that Kitchen Appliance is "abstract" (just defines the common characteristics)

- To define…. **New classes** that are added to an existing design (by extending an existing concrete class):

Thinking about extensions



Note that both Fax machine and Fax/copy machine are "concrete" classes

The most important inheritance heuristic

- The first heuristic in Chapter 5 of Arthur Riel's book:
  - H5.1: Inheritance should be used only to model a specialization hierarchy.
- H5.1 is a restatement of the Liskov Substitution Principle
  - Liskov Substitution Principle: Whenever you define a subtype, you should be able to safely substitute an object of the supertype with an object of the subtype.
  - In other words, although derived classes might have "extra" behavior, they must also implement the **full set** of base class behaviors.
- This principle is sometime called the "is-a" rule...
- This is a very important heuristic, because it affects other software designers that may want to add to an existing inheritance hierarchy
  - If you violate the "is-a" rule, existing code might be broken by the addition of new subclasses

Barbara Liskov, computer science professor at MIT, inventor of the CLU programming language (with support for "data abstraction" and subtyping)

The “is-a” rule

The behavior of a subclass must conform to the superclass interface:

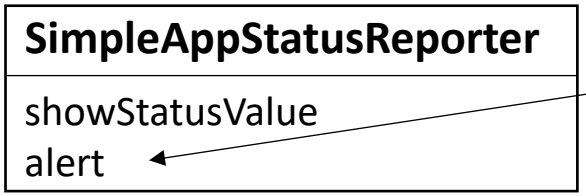
List of operations

- Each subclass has every operation that is defined in the superclass… but it is OK to add new functions to the subclass that aren’t supported by the superclass

If a subclass redefines a function, that function should:

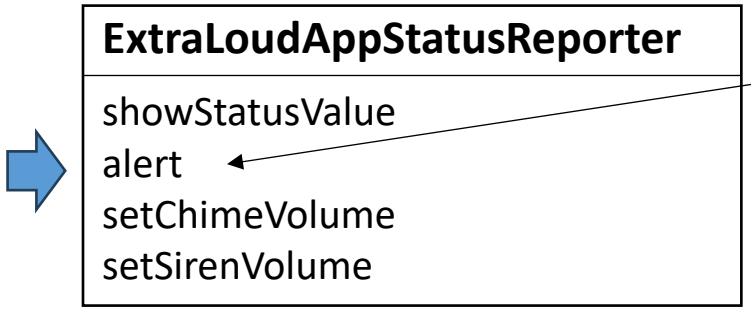
- **expect no more**
- **deliver no less**

Reports “status” of a cell phone app…



- The alert() function uses the current “volume setting”
- Postcondition: when alert() is done, phone volume setting is unchanged

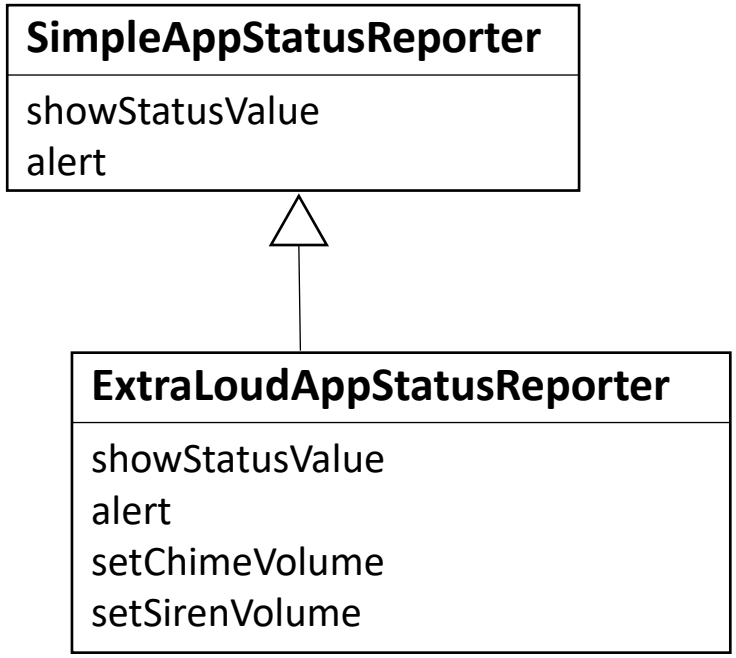
A fancier version of Status Reporter… the “alert” function is special…



- When alert() is called, the volume may be temporarily increased
- When alert() is done, phone volume setting is set back to the original volume

The “is-a” rule

Look carefully at each subclass function:



Preconditions

- The preconditions for any subclass operation are only allowed to be “weaker”
  - if a function operates on a **SimpleAppStatusReporter**, then it shouldn’t crash when we pass in an **ExtraLoudAppStatusReporter**

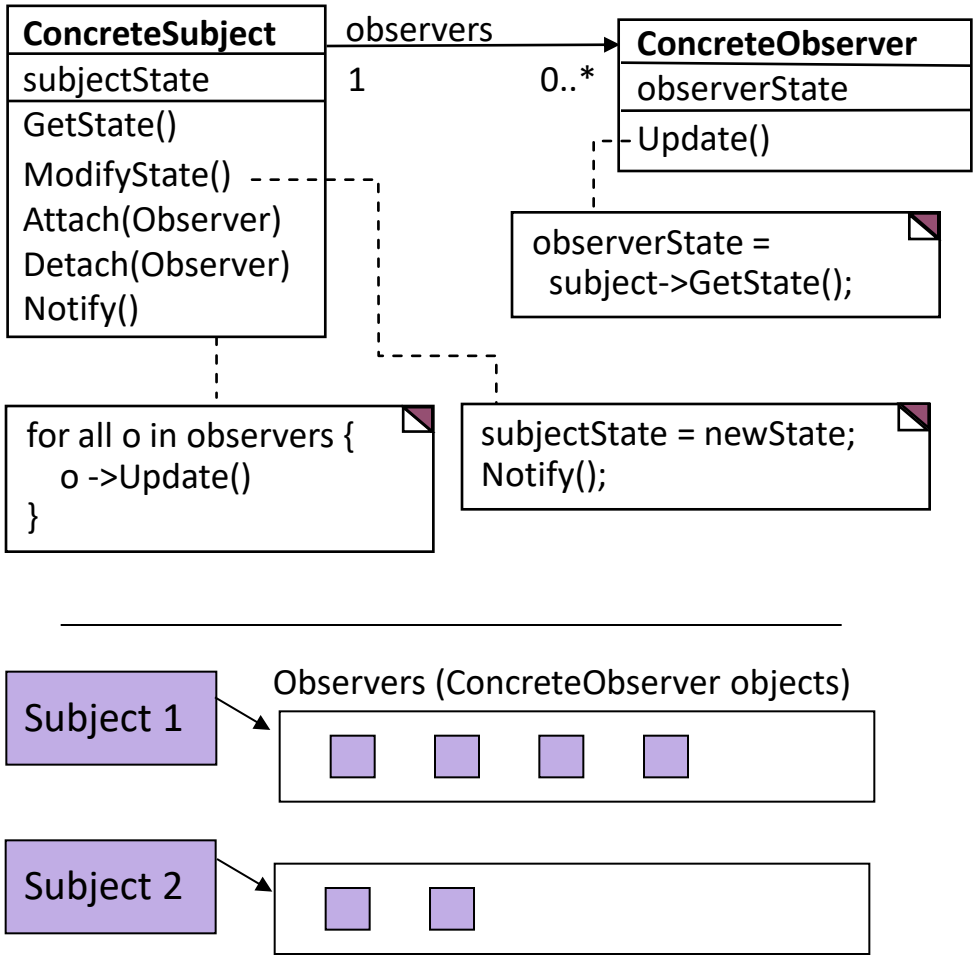
Postconditions

- The postconditions for any subclass operation are only allowed to be “stronger”
  - If **Simple** ends its “alert” operation with the device volume at the same level, then **Extra** should meet the same restriction
  - So calling “s.alert()” five times shouldn’t get progressively louder…

Observer pattern

Observer's Update function is a kind of "callback"

- Subject classes contain changing data
- Any Observer object can "register for interest" with one or more Subject objects
- Each Observer object will have its Update() operation called whenever its subject changes state



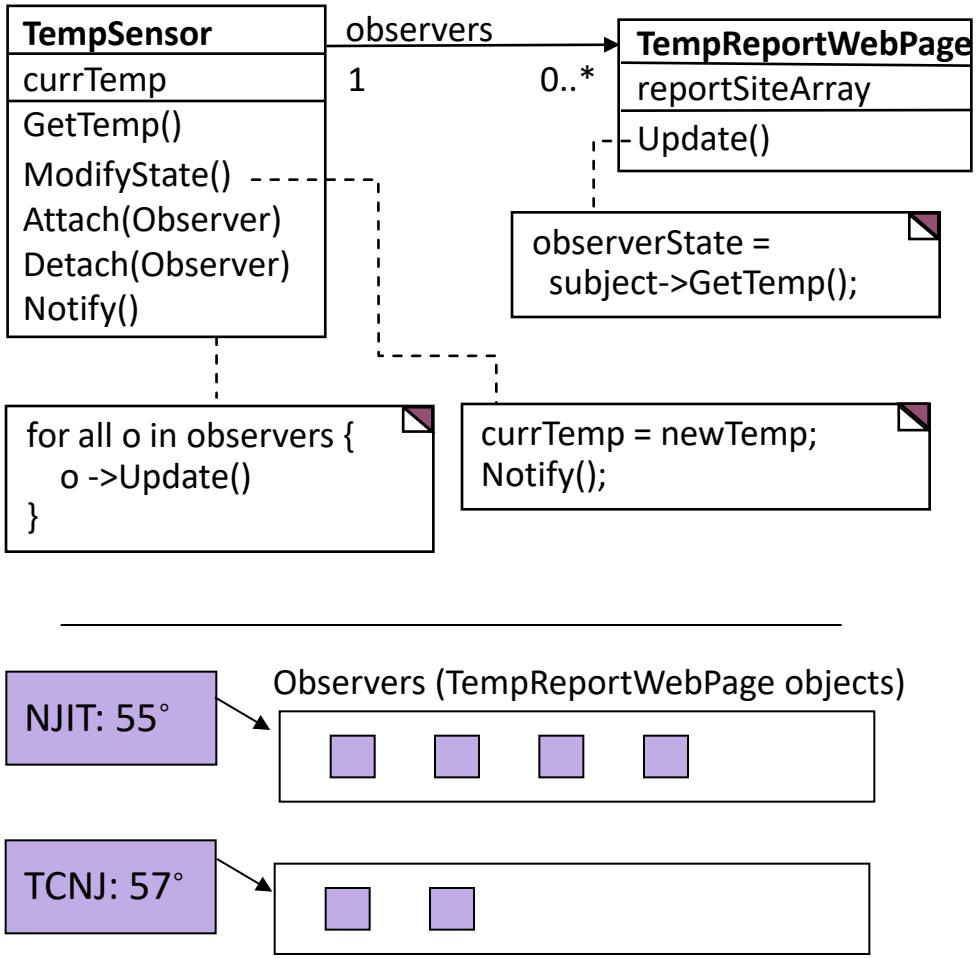
Observer pattern

Observer's Update function is a kind of "callback"

- Subject is "TempSensor" (state is currTemp)
- Any TempReportWebPage object can "register for interest" with one or more TempSensor objects
- Multiple TempReport… Update() operations called when a TempSensor changes state

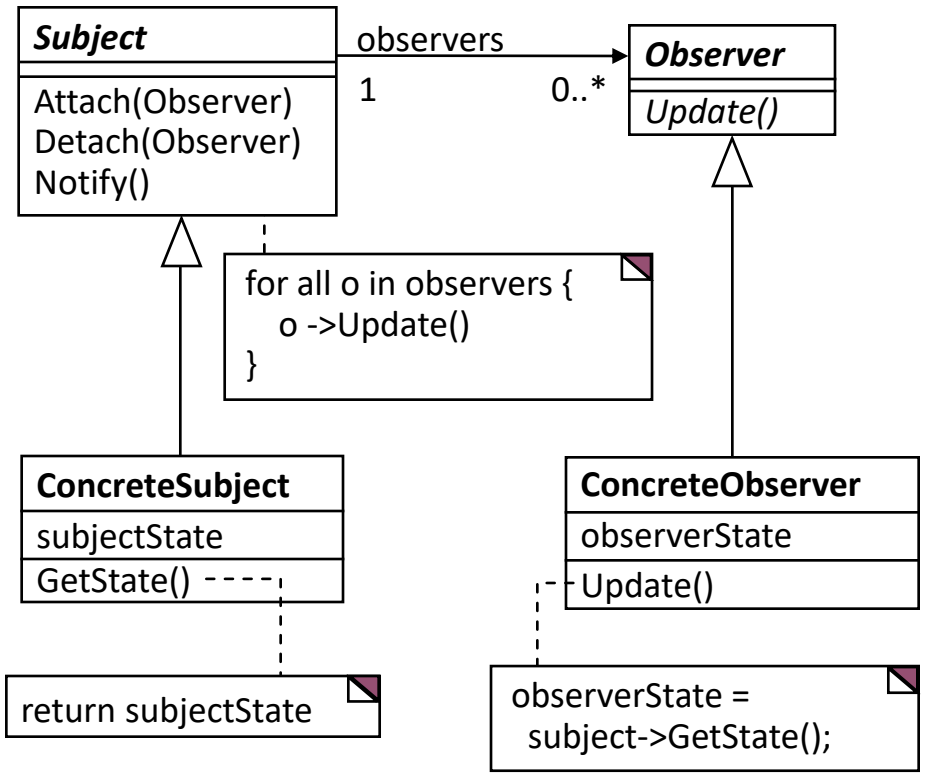
The Update() function can do many things: **trigger a real-world event**

- turn on air conditioning
- repaint a user screen
- send a message to another system



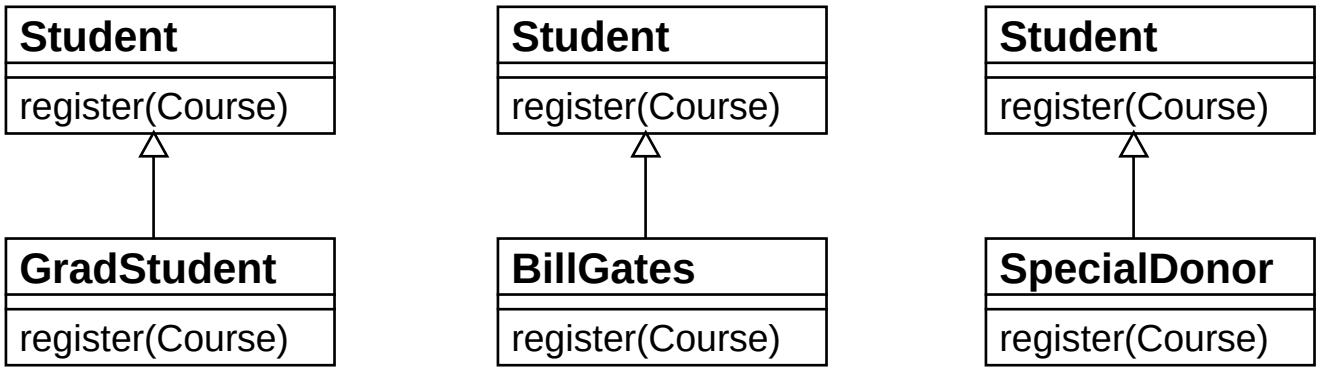
Complete class diagram for Observer

- **Observer** is an abstract interface that each **ConcreteObserver** must implement (must implement an Update() function)
- Observer objects still register by calling the Attach() operation on a **ConcreteSubject** object
- Each ConcreteObserver object will have its Update() operation called whenever its ConcreteSubject changes state



Another inheritance heuristic

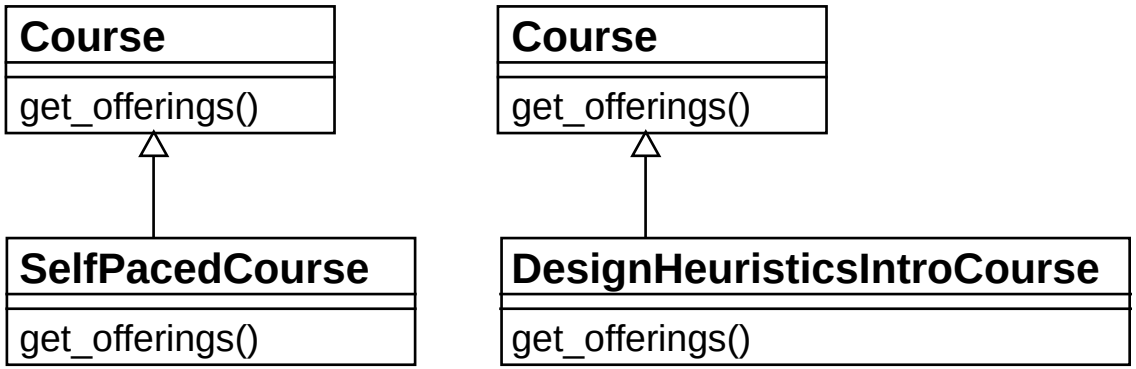
- Watch out for subclasses that are actually “singleton objects”
- Classes that are really singleton objects are usually not what we want in a simple and extensible design:



OK Not OK Probably OK

Watch for singleton objects

- Classes that are really singleton objects are usually not what we want in a simple and extensible design:



Heuristics for avoiding inheritance pitfalls

- Arthur Riel has included this in his heuristics:
  - H5.15: Do not turn objects of a class into derived classes of the class. Be very suspicious of any derived class for which there is only one instance.
- Note: this heuristics is sometimes violated for good design reasons…

  – H5.15 is sometimes violated when you need a singleton object in a framework

Summary

- What have we learned? How are we going to change the way we design and implement our software?
  - Design heuristics are useful for discussing design tradeoffs
- The main pitfalls to watch for are:
  - encapsulation problems: poorly designed modules
  - god classes: classes that steal all of the decision-making ability of the classes around them
  - improper inheritance: violations of the “is-a” rule

References

The book:

- *Object Oriented Design Heuristics* by Arthur Riel (Addison-Wesley, 1996)

Top 20 heuristics:

- http://manclswx.com/talks/top_heuristics.html

Vince Huston – listing of Arthur Riel's heuristics:

- http://www.vincehuston.org/ood/oo_design_heuristics.html

Design Principles (Bob Martin)

- http://manclswx.com/talks/Principles_and_Patterns.pdf

http://manclswx.com/talks - talks on legacy software, design patterns, technical debt, and design heuristics

Last modified: Nov. 9, 2024

Books and articles about Design Patterns

Books

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns* (Addison-Wesley, 1994)
- Eric Freeman and Elisabeth Robson, *Head First Design Patterns* (O'Reilly, 2005)
- Joshua Kerievsky, *Refactoring to Patterns* (Addison-Wesley, 2005)
- *Design Patterns in Java* (second edition), William C. Wake, Steven John Metsker (Addison-Wesley, 2006)
- Robert S. Hanmer, *Patterns for Fault Tolerant Softwar*e (Wiley, 2007)
- *Pattern Oriented Software Architecture, volume 2* by Doug Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann (Wiley, 2000)

Websites

- wiki.c2.com/?DesignPatternsBook
- hillside.net/patterns/patterns-catalog
- www.martinfowler.com/articles/enterprisePatterns.html

OO Design Heuristics: List of topics

Topics:

- Modularization
- Heuristics to help evaluate designs
- Data hiding
- Action oriented versus object oriented
- God class

More Topics:

- Using inheritance in patterns (Strategy, Observer)
- Abstract classes
- Inheritance “is-a” rule = Liskov Substitution Principle
- Expect no more, deliver no less