Object Oriented Design Heuristics

Dennis Mancl

dmancl@acm.org
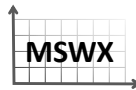
MSWX

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

Why Heuristics?

- All software developers know how to **code**, but they are not all equally skilled in **creating good designs** for applications or subsystems

- We all get training in programming languages, development tools, algorithms, data structures, and database tools

- But most of us have never had a formal course in design – we learn "on the job"

slide 2

1

Why Heuristics?

- Design principles: good but complicated…
  - <u>Modularization</u> – divide a big design into small independent pieces
  - <u>Encapsulation</u> – each component has a simple interface, hiding complex details
  - <u>Low coupling</u> – minimize dependencies between components
  - <u>Extensibility</u> – components can be extended without changing the base code

- Design heuristics: easier to learn and apply
  - Simple "rules of thumb"
  - They help avoid design pitfalls
  - Use them in design brainstorming and design reviews

Heuristics for Object Oriented Design

- We will talk about "object oriented design heuristics" today:
  - We want to define good classes (classes should encapsulate the right data and should have the right public interfaces)
  - A good distribution of responsibilities across a group of classes
    - Not too much "concentration of behavior" in one class
    - Many of our classes will "delegate" part of their work to other classes
  - Use inheritance / subtypes in a sensible way
    - Avoid major pitfalls in the use of inheritance

Design Heuristics can help

- Designers have to avoid many pitfalls
  - building an "action-oriented" system instead of an object oriented system
  - creating god classes (too much concentration of data or operations)
  - underuse of containment relationships
  - inappropriate use of inheritance to model containment, common interface, and single objects

- We will talk more about:
  - *Action oriented*: centralized control, single main procedure, uncontrolled data sharing
  - *God class*: one class controls everything, lots of complexity concentrated in one place

Arthur Riel's Object Oriented Design Heuristics

- One set of heuristics is contained in the book:

Object-Oriented Design Heuristics
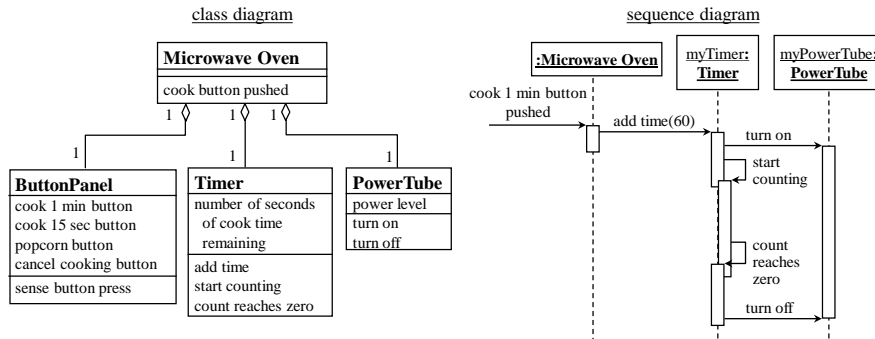### by Arthur Riel

- 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

Notation for design

- Notation
    - design will create some <u>artifacts</u> – and we want to everyone to be able to read them
    - a common set of notations is the <u>Unified Modeling Language</u> (UML), defined by Rational Software Inc. and adopted by the Object Management Group and many CASE tool vendors

class diagram

sequence diagram

Microwave Oven
cook button pushed

ButtonPanel
cook 1 min button
cook 15 sec button
popcorn button
cancel cooking button
sense button press

Timer
number of seconds of cook time remaining
add time
start counting
count reaches zero

PowerTube
power level
turn on
turn off

:Microwave Oven   myTimer**:** **Timer**   myPowerTube**:** **PowerTube**

cook 1 min button pushed

add time(60)

turn on

start counting

count reaches zero

turn off

slide 7

Categories of heuristics

- Arthur Riel's heuristics are organized into several categories:
    1. Classes and objects: the basic structure of the building blocks of OO architecture
    2. Object oriented "topology" (versus action-oriented topology): creating decentralized architectures
    3. Designing the pattern of collaborations within a system of objects
    4. The inheritance relationship: creating variations or extensions of an existing class
    5. Miscellaneous heuristics: multiple inheritance, associations, use of class data and operations, and physical design considerations

slide 8

Some important heuristics

- Arthur Riel's heuristic covers some important object oriented basics:
    - H2.1: All data should be hidden within its class.
    - H2.3: Minimize the number of messages in the protocol of a class.
    - 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.
    - H2.8: A class should capture one and only one key abstraction.
- These heuristics are the basis of many "coding standards" documents
    - In C++ and Java, most software developers make all data attributes "private" so they can only be accessed and modified by operations within the class.
    - Classes need to have simple and clean documentation.
    - Classes should avoid mixing together multiple abstractions.

Encapsulation

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

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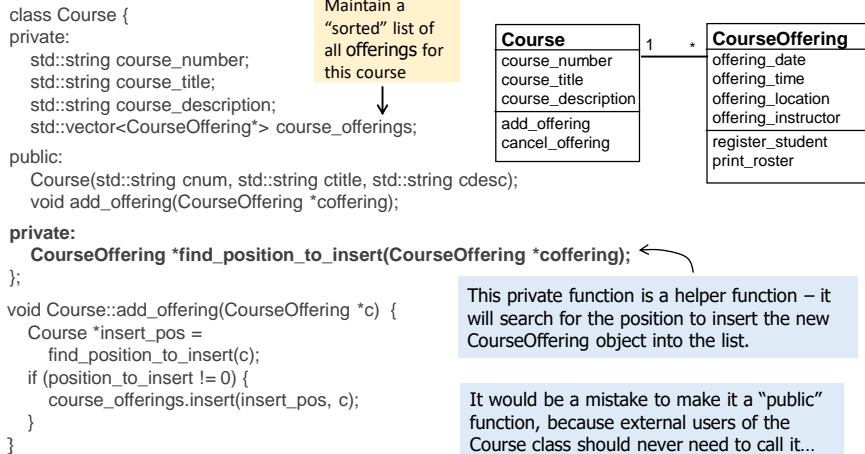
In the public section of the class, we have "accessor" and "modifier" functions to be used by others

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

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.

```
class Course {
private:
    std::string course_number;
    std::string course_title;
    std::string course_description;
    std::vector<CourseOffering*> course_offerings;
public:
    Course(std::string cnum, std::string ctitle, std::string cdesc);
    void add_offering(CourseOffering *coffering);

private:
    CourseOffering *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);
    }
}
}
```

Maintain a "sorted" list of all offerings for this course

Course	1	*	CourseOffering
course_number			offering_date
course_title			offering_time
course_description			offering_location
add_offering			offering_instructor
cancel_offering			register_student
			print_roster

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…

slide 11

Action oriented program

- What is an "action-oriented" program?
    - the application has *centralized control* – there is a single main procedure that is "in command" of the control flow
    - the application's data may be *shared* between many different procedures

- Action-oriented programs don't always evolve gracefully...
    - watch for "accidental complexity"
    - accidental complexity == complexity that is due to the structure of the implementation rather than the structure of the problem
    - when we add new functionality to an already-designed system, we often create accidental complexity

slide 12

Action oriented program

- Main program controls everything

```
/* Main program – it gives orders to all of the function */
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"
    - the design attempts to group together each important data structure and the operations that manipulate it into a single class
    - *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 <u>superficially</u> object-oriented, while still having the structural problems of an action-oriented application
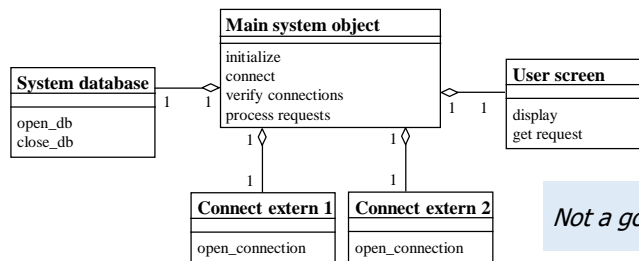
Not a good design!

Two design problems

- the "god class" problem
  - one class controls everything
- the "proliferation of classes" problem
  - the functionality is spread out over too many small classes

Some heuristics

These heuristics are from Chapter 3 of Riel's book:

- H3.1: Distribute system intelligence horizontally as uniformly as possible.

- H3.2: Do not create god classes/objects in your system.

- H3.3: Beware of classes that have many accessor methods defined in their public interface. Having many accessor methods implies that related data and behavior are not being kept in one place.

- H3.4: Beware of classes that have too much noncommunicating behavior (operations that operate on a proper subset of the data members of a class).

Central control versus object design

- These four heuristics might be violated when a designer:
  - has a centrally-controlled architecture in mind at the beginning of the design process, and
  - tries to maintain that centrally-controlled structure during the initial design of the main classes in the system.

- The designer is still <u>thinking</u> in the action-oriented paradigm, but is trying to recast the design in object-oriented terminology (without really making the transition to an object oriented architecture).

Example

- Arthur Riel's home heating system example (an object oriented design with poor distribution of behavior):
  - The system has two main physical classes: **Room** and **Furnace**
    - **Room** has three important attributes:
      - the current temperature in the room (ActualTemp)
      - the thermostat setting for the room (DesiredTemp)
      - whether the room is currently occupied (Occupancy)
    - **Furnace** can be turned on and off.
  - We introduce a new class called **HeatFlowRegulator**:
    - **HeatFlowRegulator** mediates between the **Room** and the **Furnace**.
    - It calls on the services of the **Room** class: it peeks at the values of DesiredTemp, ActualTemp, and Occupancy, performs some computation, and invokes the appropriate operations on the **Furnace**.
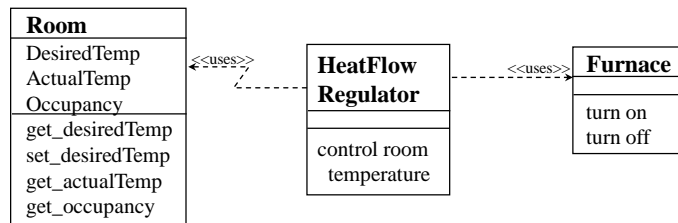
Room
DesiredTemp : double
ActualTemp : double
Occupancy : bool

Furnace
turn on
turn off

Example (continued)

- Initial ("god class") version of the home heating system

Questions about the example

- Is this system "object-oriented"?
  - Sure, because we have put all of the data and control into classes.
- Are there problems with this design?
  - Yes. The **HeatFlowRegulator** is a "god class". It is the "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.

We are going to change the "distribution of responsibilities"

The Room class will be "smarter"

Questions about the example

- Improved version of home heating system
  - Let the **Room** do the computation to determine whether it needs heat.
  - This makes the **HeatFlowRegulator** class much simpler, since it only needs to call the **Room::do_I_need_heat?( )** function (instead of all of the other accessor functions), so it doesn't depend so much on the structure of the information in **Room**.

Room
DesiredTemp
ActualTemp
Occupancy
do_I_ need_heat?

<<uses>>

HeatFlow Regulator
control room temperature

<<uses>>

Furnace
turn on
turn off
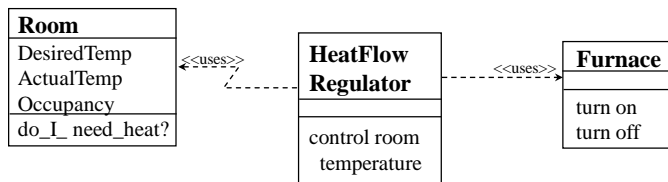
Is this a better design? What do you think?

Distribution of functionality

- The home heating system is a great illustration of Heuristic H3.1:
  - H3.1: Distribute system intelligence horizontally as uniformly as possible. This means that the top-level classes in a design should try to share their work uniformly.
- 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

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?

What is a god class?

- A "god class" indicates that there is a poor distribution of responsibility:
  - H3.2: no "god classes"
    - How can you tell if you have a god class? If a system has N classes, and you ask each developer which $N-1$ classes he or she would be willing to write, and if everyone avoids the same class – that's a hint that you have a god class.

Is a god class good or bad?

Good for building the initial design quickly?

Bad for long-term maintenance?   Difficult to implement clean modifications and extensions?

What do you think?

Heuristics to detect god classes

- Two more warning signs that you might have a god class:
  - H3.3: too many get and set operations
    - If you are writing a class A that calls several get and set functions in another class B, you might ask the question: "What am I doing with the information that I am getting from class B, and why doesn't class B do it for me?"
  - H3.4: too much noncommunicating behavior
    - This is a sign of poor cohesion (the list of operations is not well-thought-out). You may have combined two or more classes into one.

In the first design, why did HeatFlowRegulator have to call three "get" functions? Room can do the calculations instead – this makes the HeatFlowRegulator simpler.

Definition of "noncommunicating behavior" is on the next page…

Noncommunicating behavior

- Problem: a small subset of the public member functions of the class are implemented only in terms of a small subset of the data attributes

```
class CustomerOrder {
private:
    std::string customer_name;
    std::string customer_billing_address;
    std::string customer_shipping_address;
    Money order_cost;
    std::vector<std::string> order_item_names;
    std::vector<Money> order_item_costs;

public:
    void set_customer_info(std::string name,
        std::string addr1, std::string addr2);
    void print_mailing_label() const;
    void add_new_item(std::string item_name, Money item_cost);
    void clear_all_items();
    Money get_cost() const;
};
```

A CustomerOrder record is an object that contains all of the information needed to prepare an order for shipment to a customer
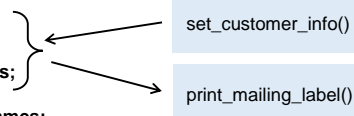
Noncommunicating behavior

- Problem: a small subset of the public member functions of the class are implemented only in terms of a small subset of the data attributes

```
class CustomerOrder {
private:
    std::string customer_name;
    std::string customer_billing_address;
    std::string customer_shipping_address;
    Money order_cost;
    std::vector<std::string> order_item_names;
    std::vector<Money> order_item_costs;

public:
    void set_customer_info(std::string name,
        std::string addr1, std::string addr2);
    void print_mailing_label() const;
    ....
};
```

set_customer_info()

print_mailing_label()

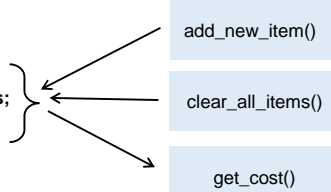*Note: these two functions only act on the "customer" information, not the "item" or "cost" information*

Noncommunicating behavior

- Problem: a small subset of the public member functions of the class are implemented only in terms of a small subset of the data attributes

```
class CustomerOrder {
private:
  std::string customer_name;
  std::string customer_billing_address;
  std::string customer_shipping_address;
  Money order_cost;
  std::vector<std::string> order_item_names;
  std::vector<Money> order_item_costs;

public:
 ....
  void add_new_item(std::string item_name, Money item_cost);
  void clear_all_items();
  Money get_cost() const;
};
```

add_new_item()

clear_all_items()

get_cost()

Note: these three functions only act on the "item" and "cost" information

How to fix non-communicating behavior: make smaller classes

- A Solution: The design can be improved by creating two new classes:

```
class Customer {
private:
  std::string name;
  std::string billing_address;
  std::string shipping_address;
public:
  Customer(std::string cname,
     std::string cbilladdr,
     std::string cmailaddr);
  void print_mailing_label() const;
};

class Item {
private:
  std::string name;
  Money cost;
public:
  Item(std::string iname, Money icost);
  Money get_item_cost() const;
};
```

- The new CustomerOrder class will now contain a Customer (by reference) and Items (by value):

```
class CustomerOrder {
private:
  Customer *cust;
  std::vector<Item> items;
  Money order_cost;

public:
  CustomerOrder(Customer *c);
  void add_new_item(const Item &it);
  void clear_all_items();
  Money get_cost() const;
};
```

Customer and Item are helper classes – do they make the design simpler and better? What do you think?
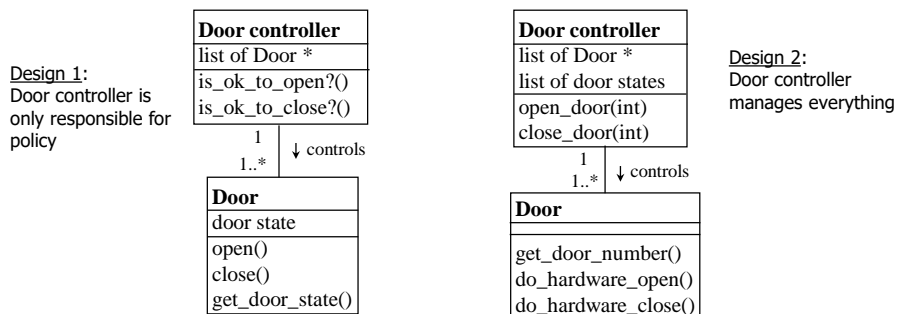
When to violate the noncommunicating behavior heuristic

- This is a heuristic – <u>not</u> a rule
- Some classes just have a large number of attributes
  - but there are no logical classes that can be split off
- Creating many small classes can create performance problems in some environments
  - for example, Java classes that need to be downloaded from the Web
- When a designer decides to use a wrapper class approach, it is common to define a single class that describes a particular "interface" that communicates with other subsystems
  - this technique is commonly used with various forms of "component technology"

slide 29

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 be the doors of a subway train, the doors of a supermarket, or the doors in a secure building.

<u>Design 1</u>:
Door controller is only responsible for policy

Door controller
list of Door *
is_ok_to_open?()
is_ok_to_close?()

1
1..* ↓ controls

Door
door state
open()
close()
get_door_state()

Door controller
list of Door *
list of door states
open_door(int)
close_door(int)

1
1..* ↓ controls

Door
get_door_number()
do_hardware_open()
do_hardware_close()

<u>Design 2</u>:
Door controller manages everything

slide 30

15

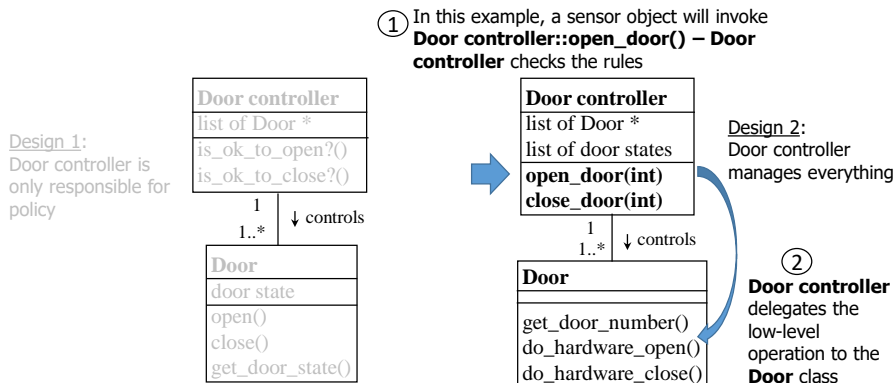# Example: Design of a Door system

- 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
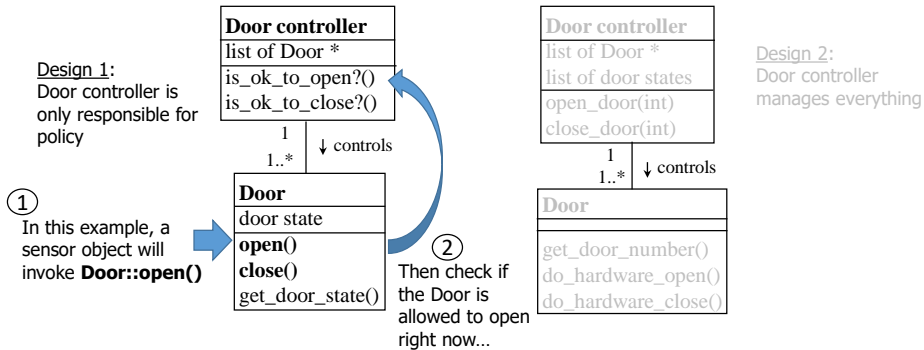
slide 31

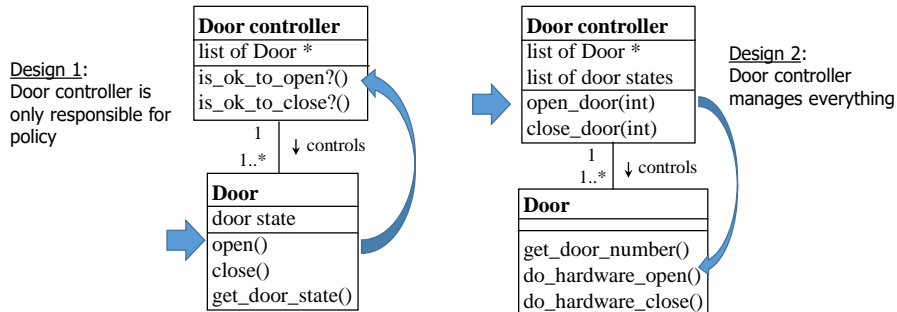# Example: Design of a Door system

- 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

slide 32

Example: Design of a Door system

- 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

Door controller

| list of Door * |
| is_ok_to_open?() |
| is_ok_to_close?() |

1
1..*   ↓ controls

Door

| door state |
| open() |
| close() |
| get_door_state() |

Design 2:
Door controller manages everything

Door controller

| list of Door * |
| list of door states |
| open_door(int) |
| close_door(int) |

1
1..*   ↓ controls

Door

| get_door_number() |
| do_hardware_open() |
| do_hardware_close() |

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

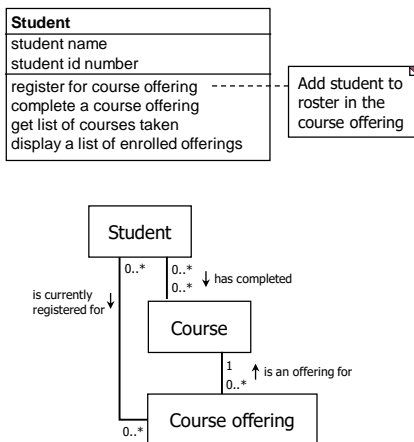| Design 1 = more flexible; easier to extend | Design 2 = more secure; better performance |

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

Collaboration between classes

Student
student name
student id number
register for course offering
complete a course offering
get list of courses taken
display a list of enrolled offerings

Add student to roster in the course offering

Student
0..* 0..* has completed
0..*
is currently registered for
Course
1 is an offering for
0..*
Course offering
0..*

Student is the class that represents the information within the system contains relating to a specific student.
- How many Student objects will there be in the system??

The operations in the Student class will execute some of the scenarios are triggered by a human user of the registration system:
- Register for a course offering
- Get information about current and past courses

In order to execute these scenarios, a Student object will need to collaborate with other objects in the system: Course and Course offering objects.
- How do we find the right objects?

How to find the object to collaborate with

- Navigation within an object-oriented system
- How do you know which object to send a message to?
  1. the object may be a local object (an embedded attribute in the current object) [Container class]
  2. the object may be a parameter that was passed into the currently running operation (by name, value, or reference)
  3. the object may be found by requesting it from another object (for example, finding it in a map or within another data structure that the object knows about) [Special navigation object]
  4. the object may be a global object
  5. you might create a new local object to send the message to [Temporary object]
  6. the object may be pointed to by a local referential attribute [Association]
- Each one of these situations may occur in an object oriented design

slide 37

Six ways to collaborate

- Example of the six ways to "find the collaborating object" (from Arthur Riel's book):

$\Rightarrow$ Suppose you are a Car object and you want to collaborate with a Gas station to call the give_gasoline() operation:

1. **local embedded object** (a mini-gas station attribute within the current object – such as a "reserve gas tank")

2. **passed as a parameter** (someone called the Car's get_gasoline() function and passed as an argument the name and location of a gas station)

3. **requesting it from another object** (you have the name of the station, and you look it up in a data structure class to get an address or pointer)

Pick up the kids, and buy some gas at the Exxon station on First St.

Where is the closest gas station?

slide 38

19

Six ways to collaborate

- Example of the six ways to "find the collaborating object" (from Arthur Riel's book):

$\Rightarrow$ Suppose you are a Car object and you want to collaborate with a Gas station to call the give_gasoline() operation:

4. **global object** (only one gas station, and everyone knows where it is)
5. **create a new local object** (whenever you need gas, you build a gas station on the spot, get gasoline from it, and destroy it when you're done)
6. **a local referential attribute** (when the car is built or sold, it contains a pointer that has been initialized to the right gas station to go to for gas – the car dealer gives you a discount card for a specific local gas station)

slide 39

Collaboration examples

- Sending a message to a local object
  - a **Microwave Oven** sets its **Timer** to 1 minute (and **Timer** is contained within the **Microwave Oven**)
- Asking another object to find the right object to collaborate with
  - an **Automated Teller Machine** asks the central bank for a **Customer record** so it can verify the id number
- Creating a new local object to collaborate with
  - a **User interface** object may create a temporary **Dialog box** object to display a warning message
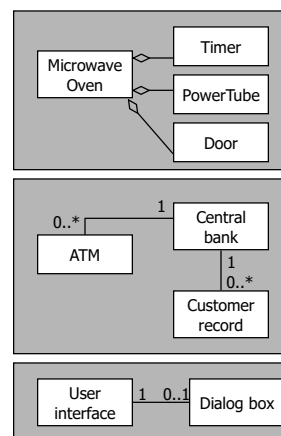
slide 40

Some collaboration heuristics

These heuristics are from Chapter 4 of Arthur Riel's book:

- H4.1: Minimize the number of classes with which another class collaborates.

- H4.2: Minimize the number of message sends between a class and its collaborator.

- H4.3: Minimize the amount of collaboration between a class and its collaborator, that is, the number of different messages sent.

- H4.4: Minimize the fanout of a class, that is, the product of the number of messages defined by the class and the messages they send.

Riel makes two observations about this set of heuristics:

- Heuristic H4.1 is the most important of the four: the main factor in the "complexity" of a class is the number of other classes it needs to use

- It is silly to set absolute limits for each of these metrics: it is better to exercising good judgment rather than blindly following an absolute rule
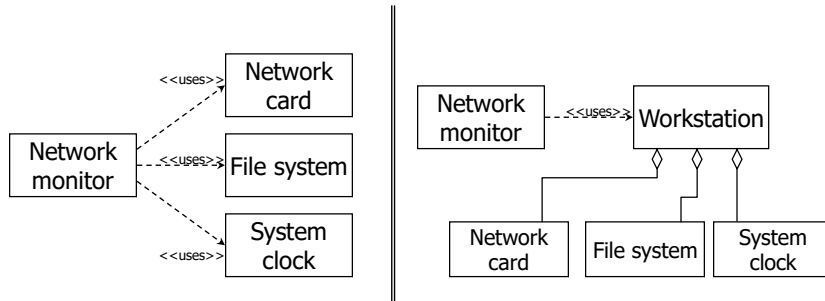
slide 41

Controlling complexity

- We use these four heuristics to help make the design simpler…

- These four heuristics might be violated when a designer:
    - creates mega-classes with large numbers of unrelated responsibilities
    - creates a behavioral god class, which requires other classes to abdicate their responsibilities to the central class
    - breaks up a class too far, so that many of the algorithms require continually requesting information from another object

- The designer is trying to create a set of cooperating classes, but the set of services offered by each class may be poorly coordinated with the other classes in the design.
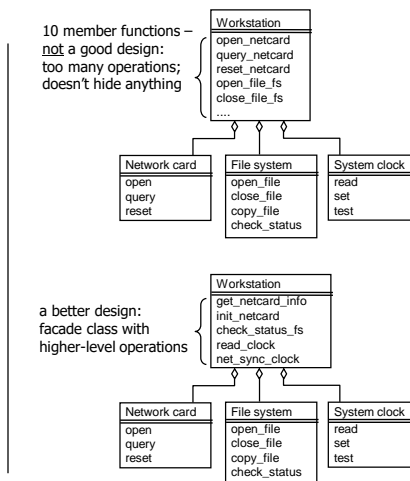
slide 42

Using containment

- One way to simplify the "uses" ("collaborates with") relationships in a design is to combine several objects into a single aggregate object:

Containment plus some abstraction

- If you create an aggregate object in your design, you might put all of the public operations in the contained classes into the aggregate object
    - not a good practice
    - we are providing too much detailed information to other classes – there is a good chance that we might create a god class
- A better idea is to do a little bit of abstraction
    - the only public functions in the **Workstation** class will be the information that needs to be fed to the **Network monitor**
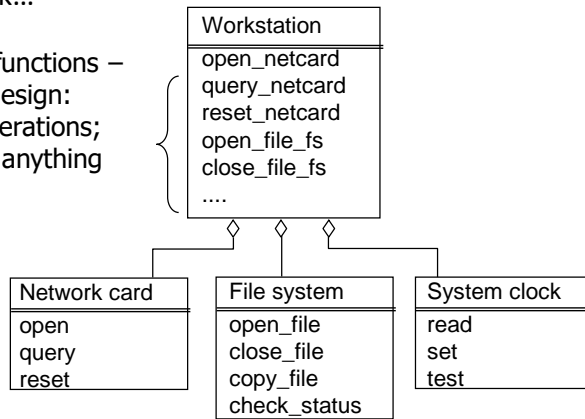- This is the Facade pattern (from the *Design Patterns* book)

Facade class public interface should "raise the level of abstraction" – don't repeat all primitive operations

Containment plus some abstraction

• A closer look…

10 member functions –
<u>not</u> a good design:
too many operations;
doesn't hide anything
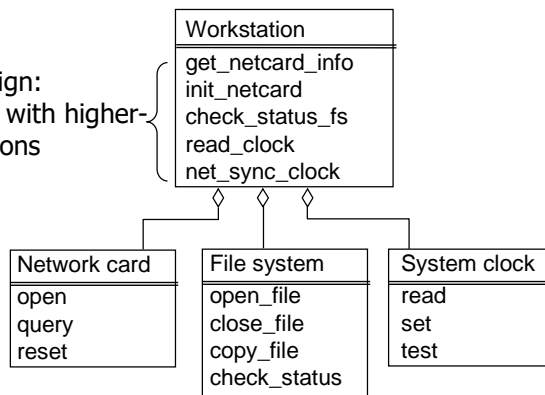
Workstation
open_netcard
query_netcard
reset_netcard
open_file_fs
close_file_fs
....

Network card
open
query
reset

File system
open_file
close_file
copy_file
check_status

System clock
read
set
test

slide 45

Containment plus some abstraction

• A closer look…

a better design:
facade class with higher-
level operations

Workstation
get_netcard_info
init_netcard
check_status_fs
read_clock
net_sync_clock

Network card
open
query
reset

File system
open_file
close_file
copy_file
check_status

System clock
read
set
test

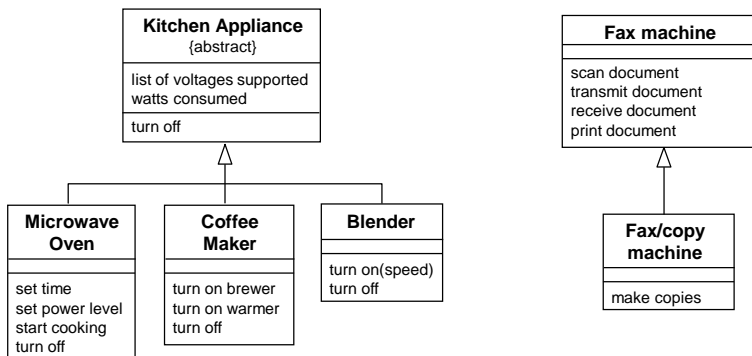slide 46

Heuristics related to containment

- Here are two more collaboration heuristics:
  - H4.5: If a class contains objects of another class, then the containing class should be sending messages to the contained objects, that is, the containment relationship should always imply a uses relationship.
  - H4.7: Classes should not contain more objects than a developer can fit in his or her short-term memory. A favorite value for this number is six.

H4.5: uses relationship is implied by containment

H4.7: add a level in the containment hierarchy

Workstation

<<uses>>

Network card   File system   System clock

LogMsg

Time | Date | Port | Link | Dest | Code | Text | Prio

LogMsg

MsgTime   MsgLocation   MsgReason

Time | Date | Port | Link | Dest | Code | Text | Prio
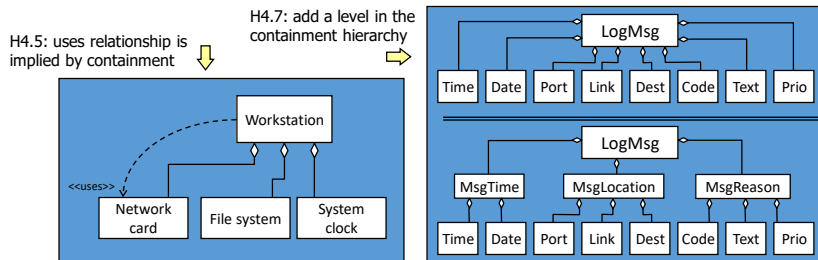
slide 47

Two ways to use inheritance

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

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

Kitchen Appliance
{abstract}

list of voltages supported
watts consumed

turn off

Microwave Oven

set time
set power level
start cooking
turn off

Coffee Maker

turn on brewer
turn on warmer
turn off

Blender

turn on(speed)
turn off

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

Fax machine

scan document
transmit document
receive document
print document

Fax/copy machine

make copies

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

slide 48

24

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 <u>substitute</u> 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 must implement 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

SimpleAppStatusReporter
showStatusValue alert

ExtraLoudAppStatusReporter
showStatusValue alert setChimeVolume setSirenVolume

Preconditions
- The preconditions for any subclass operation are only allowed to be "weaker"
  - we don't want to break existing code: 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…

Smiley face – one way to check a subclass

For each function in the subclass that is a "redefinition"
of the superclass function:
- compare the preconditions for the subclass function
  and the superclass function
  - wider means "more permissive precondition"
- compare the postconditions
  - narrower means "more restrictive postcondition"

inputs ➡
↓
outputs ➡
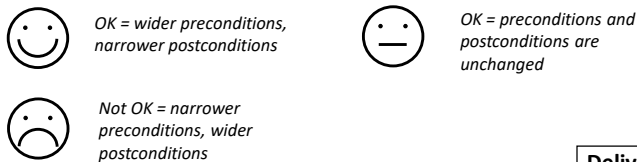
*More permissive =
wider*

*More restrictive =
narrower*
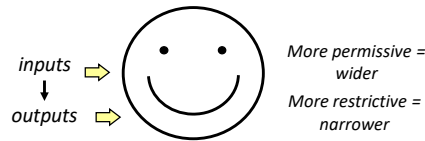
Source: Elisa Banniassad, "Making the Liskov Substitution Principle Happy and Sad,"
https://2017.splashcon.org/event/splash-2017-splash-e-making-the-liskov-
substitution-principle-happy-and-sad

Smiley face – one way to check a subclass

Let's do an example – can we substitute a
SkateboardDeliveryPerson for a DeliveryPerson?

*OK = wider preconditions,
narrower postconditions*

*OK = preconditions and
postconditions are
unchanged*

*Not OK = narrower
preconditions, wider
postconditions*

reportLocation

*For a Skateboard delivery, the
reportLocation preconditions and
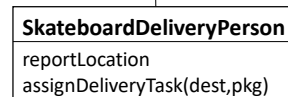postconditions are unchanged*

assignDeliveryTask

*For a Skateboard delivery,
assignDeliveryTask precondition is
narrower:*
- *5 pound weight limit for package*
- *Address must be within 3 miles*

DeliveryPerson
reportLocation
assignDeliveryTask(dest,pkg)

??

SkateboardDeliveryPerson
reportLocation
assignDeliveryTask(dest,pkg)
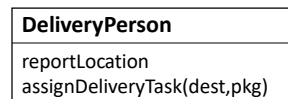
Well… maybe this isn't
a good subclass!

Violations of the is-a rule

- The most common violation of H5.1 is the use of inheritance to model a "has-a" relationship instead of an "is-a" relationship

```
public class Customer {
    private String cust_name;
    ....
}

public class CustomerOrder extends Customer {
    // this permits the Customer.getname() operation
    // to be invoked on a CustomerOrder object
    //  --- this is *not* a good design!! ---
    ....
}
```

> Java example:
> Notice that you won't get a compiler error for the CustomerOrder class. It is a "design-level" problem.

Complexity and inheritance depth

- Some thoughts on inheritance depth:
  - H5.4: In theory, inheritance hierarchies should be deep – the deeper the better.
  - H5.5: In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six.
- This isn't an absolute rule: the warning flags should go up at six…

> Arthur Riel's explanation for Heuristic 5.5 (page 84):
> "Developers get lost in the levels if the hierarchy is too deep. This problem can be partly alleviated with support from tools (tools that let you see the entire public interface from a class, including inherited operations)."

> The problem with deeply-nested class hierarchies:
> - there can be a big semantic difference between objects at different levels of the tree
> - subclasses may have special internal states and complicated rules

Inheritance pitfalls

- One inheritance pitfall – defining several derived classes that are actually "states" of the main class:

```
Stack
stack_pointer
push
```

EmptyStack   NonEmptyStack
                pop
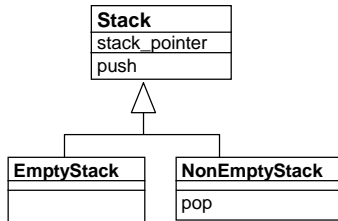
⇒ A **Stack** object gets created as a **EmptyStack** (so you can't pop it), but it becomes a **NonEmptyStack** after the first element is added – this is not a good design!

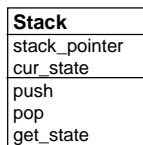⇒ The problem is: in most cases, an object should never change its class.

Stack stack1 = new EmptyStack;

(empty)

stack1.push(100);
stack1.push(200);
stack1.push(300);

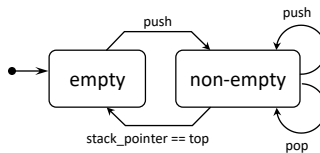How did stack1 turn into a NonEmptyStack?;

```
300
200
100
```

Correct way to model a stateful class

- The **Stack** class hierarchy can be collapsed into a single class by including a state model:

class model

```
Stack
stack_pointer
cur_state
push
pop
get_state
```

state model

push

empty   non-empty   push

stack_pointer == top   pop

⇒ The state model can show which operations are <u>legal</u> and which are <u>illegal</u> at various points in the object's lifecycle
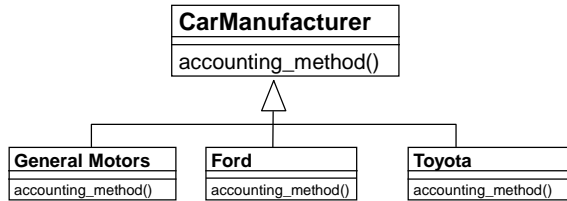
```
/* one possible implementation of the pop() operation */
int Stack::pop() {
  int val = -1;
  if (cur_state == empty) {
      // do nothing, set an error code, or throw an exception
  }
  else { stack_pointer--;  val = *stack_pointer;  }
  if (stack_pointer == top) {  cur_state = empty;  }
  return (val);
}
```

Another inheritance pitfall

- Defining derived classes that have only one instance:

CarManufacturer
accounting_method()

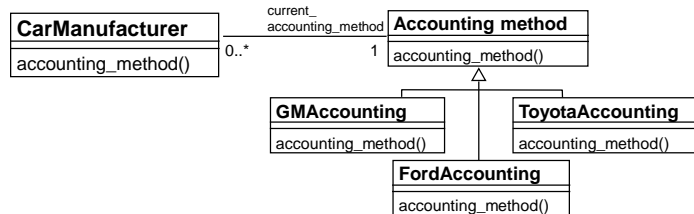| **General Motors**	**Ford**	**Toyota**
accounting_method()	accounting_method()	accounting_method()

$\Rightarrow$ each class can only have one instance, so each class is
not very reusable... it would be better to make the
base class accounting_method() operation collaborate
with an **AccountingMethod** class hierarchy

Another inheritance pitfall

- The Accounting method behavior may be factored into a separate class,
so you don't need subclasses of **CarManufacturer**:

CarManufacturer
accounting_method()

current_
accounting_method

0..*   1

Accounting method
accounting_method()

GMAccounting	**ToyotaAccounting**
accounting_method()	accounting_method()

FordAccounting
accounting_method()

Watch for singleton objects

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

Student
register(Course)

△

GradStudent
register(Course)

OK

Student
register(Course)

△

BillGates
register(Course)

Not OK

Student
register(Course)

△

SpecialDonor
register(Course)

Probably OK
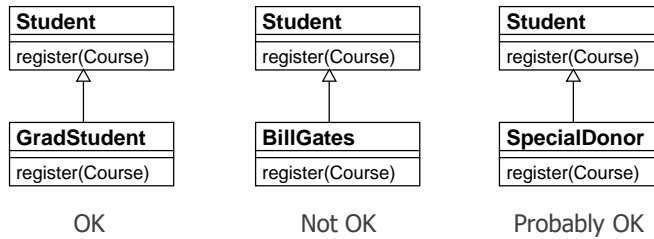
slide 59

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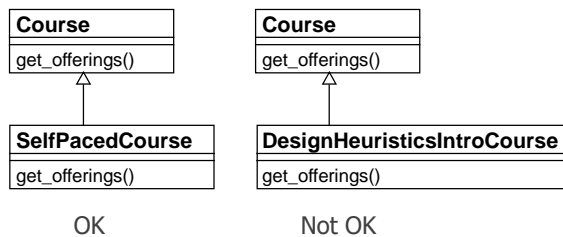
- Classes that are really singleton objects are usually not what we want in a simple and extensible design:

Course
get_offerings()

△

SelfPacedCourse
get_offerings()

OK

Course
get_offerings()

△

DesignHeuristicsIntroCourse
get_offerings()

Not OK

slide 60

Heuristics for avoiding inheritance pitfalls

- Arthur Riel has turned these two problems into heuristics:
  - H5.14: Do not model the dynamic semantics of a class through the use of the inheritance relationship. An attempt to model dynamic semantics with a static semantic relationship will lead to a toggling of types at runtime.
  - 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: these two heuristics are sometimes violated for good design reasons…

 - H5.14 may be violated in the "virtual constructor" idiom:
    - a "factory method" creates new objects
    - if the factory method is creating objects that have been stored in a file, it might create an object with an initial interim datatype and then transform the object to the correct datatype when all of the data section is complete

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

slide 61

Avoid defining derived classes with NOP operations

- If a subclass satisfies the "is-a" relationship, it must provide meaningful functionality for each operation in the superclass
  - either the subclass inherits the implementation of an operation in the superclass
  - or the subclass provides a new definition of the operation that is defined in the superclass
  - but the subclass should not redefine the superclass as a "null operation"
- This is yet another inheritance heuristic:
  - H5.17: It should be illegal for a derived class to override a base class method with a NOP method, i.e. a method which does nothing.
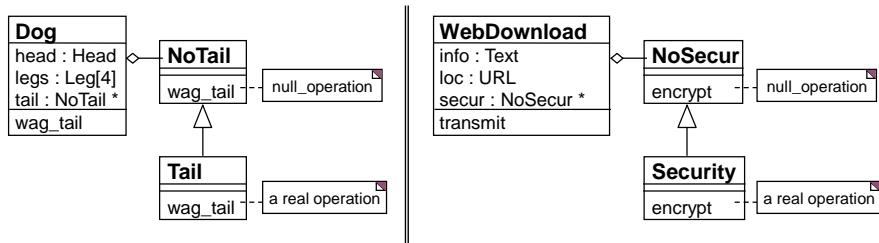

Dog
tail
bark
wag_tail

DogNoWag
wag_tail
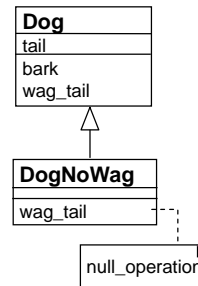
null_operation

poor design:
- canceling an operation in the base class
- we don't "deliver the complete functionality of the base class"

slide 62

One solution: design a class with optional parts

- A possible solution to the "NOP" problem is to model objects that have optional parts
  - this kind of model uses containment in addition to inheritance
  - the model can associate some special behavior with the optional parts, but the main behavior can be in the main class

Dog
head : Head
legs : Leg[4]
tail : NoTail *
wag_tail

NoTail
wag_tail

null_operation

Tail
wag_tail

a real operation

WebDownload
info : Text
loc : URL
secur : NoSecur *
transmit

NoSecur
encrypt

null_operation

Security
encrypt

a real operation

Summary

- What have we learned? How are we going to change the way we design and implement our software?
  - C++ and Java programming guidelines are OK, but they are not enough to assure software quality
    - it is easy to write superficially object oriented software
    - but we need to apply some of the design heuristics
- The main pitfalls to watch for are:
  - god classes: classes that steal all of the decision-making ability of the classes around them
  - combining several classes into one: look for non-communicating behavior and find opportunities to make more cohesive classes
  - complex collaborations: breaking up the responsibilities too far can create a maintenance headache
  - improper inheritance: violations of the "is-a" rule

Improving your designs

- This course has presented a number of object oriented design heuristics: they are simple ideas that can help you make design decisions
    - use them during the process of creating new designs (when you are originally designing the classes for your system)
    - use them during design reviews (when you are discussing your designs with the rest of your team)
- These design heuristics are <u>not</u> hard-and-fast rules
    - these heuristics are occasionally violated in good designs
    - some of the heuristics are contradictory
- The design heuristics are one more tool in your design toolkit

slide 65

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://mil-oss.org/resources/objectmentor_design-principles-and-design-patterns.pdf
- (also at http://manclswx.com/talks/Principles_and_Patterns.pdf)
- (originally at the old ObjectMentor site – now deleted – http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)

Last modified: November 29, 2017

slide 66