Type Design: Part I
Types and Responsibility Allocation
Second Model

We have our design input… now what?

- We want:
  - suitable objects
  - in the right place
  - receiving appropriate messages from other appropriate objects
    - need low coupling
      - few arguments
      - number of client objects should be small
    - high cohession
      - however if accept a client object,the client object can and perhaps should send a lot of messages
- Objects need
  - Behaviour → methods (code)
  - State → instance variables

2

Objects Not Classes

- Instances versus classes
  - Our design begins with the most important things – objects (not classes)
- Entities and instances
  - Classes are arbitrary, complex and very varied
    - It's unlikely they're going to be hinted at by the analysis
  - And they're not
    - The analysis entities hint at object instances

3

… Objects Not Entities

- Entities versus instances
  - While entities suggest instances
    - I.e. there is correspondence in *identity*
  - And while entities' attributes suggest query services
    - I.e. there is correspondence in micro-state
  - There are differences, important differences
    - Entities aren't opaquely encapsulated; objects are
    - Entities don't communicate exclusively by messages; objects do
    - Instances tend not to carry out the same processing as their entity counterparts (if they carry out any processing at all)
    - I.e. outside the various manifestations of state, there is much less correspondence

4

Outside-In Design

- Naïve design
  - Expecting to work exclusively with boxes in analysis diagrams, and expecting those boxes to lead to good classes, is naïve
  - Instead we need to work with instance diagrams as well (sequence diagrams), and to expect the analysis to lead us to instances
- Only paying lip service to opaque encapsulation
  - If an object instance will be opaquely encapsulated, it's naïve and pointless to start object design with their insides!

5

…Outside-In Design…

Design Goals (order is important):

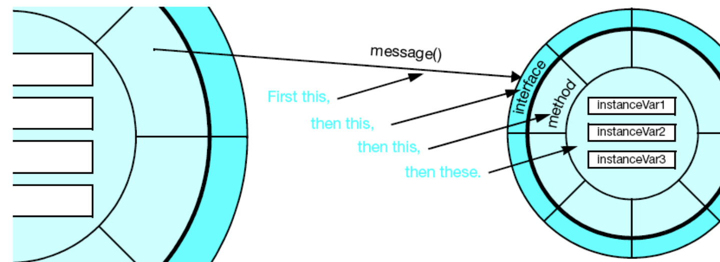
- Cohesive and loosely-coupled objects
- in the right place
- exhibiting the right interfaces
- to the right clients
- defined/made by cohesive, loosely-coupled classes
- holding the right mixture of cohesive and loosely-coupled methods
- supported by sensible instance variables
- normalized via self-messaging
- normalized via component objects
- perhaps normalized via implementation inheritance

normalized means that a feature must be presented:

- in one place
- fully at that place
- and only at that place

6

… Outside-In Design



We will

- Start with instances
- Start with their outsides – the message they can be expected to accept
- That will lead us to implementation types such as pABCs or interfaces
- And then we will be led us to the outsides of classes
- And that will lead us to public methods (member functions)
- And that will lead us to non-public methods
- And the methods will lead us to instance variables (non-static data members)
- (And only then will be led to implementation inheritance)

7

Messages

- A request by a client object
  - request can be
    - informational
      - both giving and receiving
    - behavioral
      - asks the object to do something
    - both
- Can bring information with it
  - message arguments
    - can be other objects, including the client itself
- Can return information to the client
  - there is always a return value
    - can be a primitive data type or an object
- Has a signature
  - message name + argument types + return value type
  - messageName(arg1:type1, arg2:type2, …):type

8

Interfaces / Implementation Types

- Type definition in programming:
  - a name describing the kind of data being stored
  - but in oo design data is hidden…shouldn't be named
- Type definition in oo design
  - a name describing the set of messages that an object can receive

9

Interfaces / Implementation Types

- Types can be built from other types
  - Union of message sets
  - In Java, union must be stated explicitly with *extends class* or *implements interface*
- Types ≠ Classes
  - all classes have a type
    - but a class can be associated with many different types (see above)
  - not all types are implemented as a class
    - in Java *abstract classes* and *interfaces* have incomplete implementation or none at all
    - Objects cannot be formed by such types, but they are still very useful

10

Interfaces / Implementation Types

- Types allow you to write generalized code - i.e. reuse
  - should know your types before you code your methods (i.e. implement your messages)
  - use the most general type you can
    - a meaningful type that contains the smallest set of messages
  - another word for this is **polymorphism**

11

Polymorphism, Interfaces and Abstract Classes in Java

A slight digression…

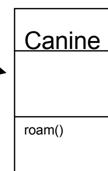
Introduction to Polymorphism

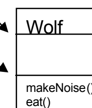
- Inheritance:
  - base class defined, and other classes derived from it
  - Code for the base class used for base objects, as well as objects of any derived classes
- Polymorphism:
  - changes made to method definitions in the derived classes
  - *changes apply to the software written for the base class*

13

Which Method is called?

```
Animal w = new Wolf();
w.makeNoise();
w.roam();
w.eat();
w.sleep();
```





The lowest one in the hierarchy wins.

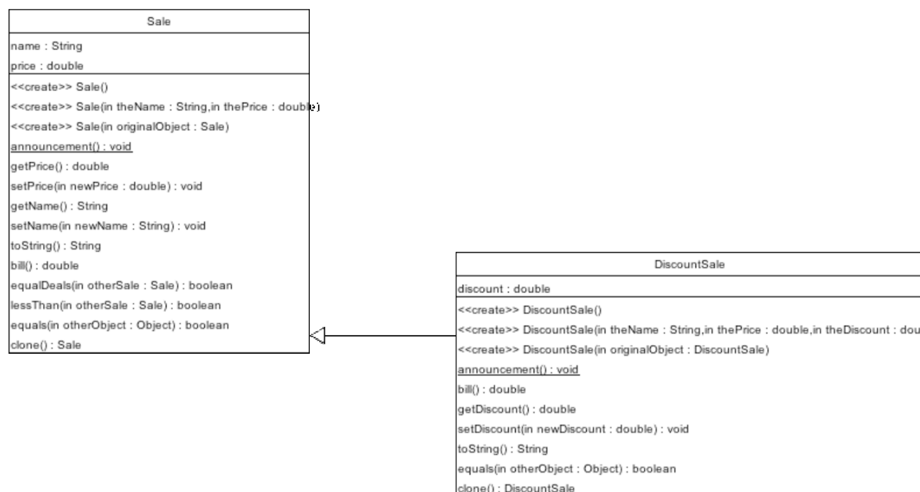
14

Late Binding

- Polymorphism works because of late binding
- associating a method definition with a method invocation is called *binding*
- If association happens when the code is compiled, that is called *early binding*
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called *late binding* or *dynamic binding*
- Java uses late binding for all methods
  (except private, **`final`,** and static methods)

15

Sale and Discount Sale



16

Sale and DiscountSale

- The **Sale** class **lessThan** method
  - Note the **bill()** method invocations:

```
public boolean lessThan (Sale otherSale)
{
  if (otherSale == null)
  {
    System.out.println("Error: null object");
    System.exit(0);
  }
  return (bill( ) < otherSale.bill( ));
}
```

17

Sale and DiscountSale

- The **Sale** class **bill()** method:

```
public double bill( )
{
  return price;
}
```

- The **DiscountSale** class **bill()** method:

```
public double bill( )
{
  double fraction = discount/100;
  return (1 - fraction) * getPrice( );
}
```

18

`Sale` and `DiscountSale`

- Given the following in a program:

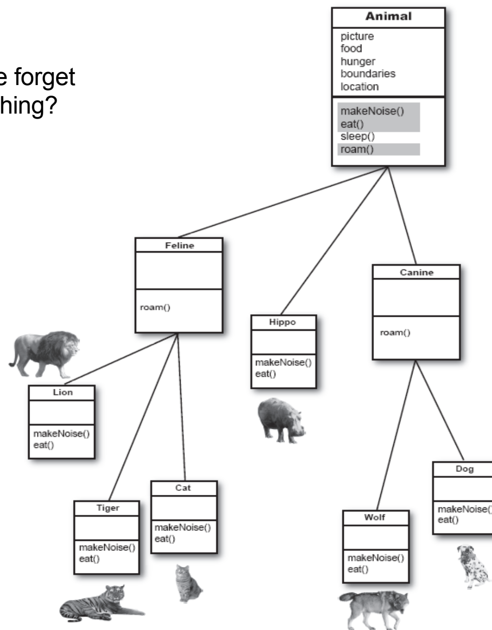
```
  . . .
Sale simple = new sale("floor mat", 10.00);
DiscountSale discount = new
           DiscountSale("floor mat", 11.00, 10);
  . . .
if (discount.lessThan(simple))
  System.out.println("$" + discount.bill() +
                  " < " + "$" + simple.bill() +
                  " because late-binding works!");
  . . .
```

  – Output would be:

```
$9.90 < $10 because late-binding works!
```

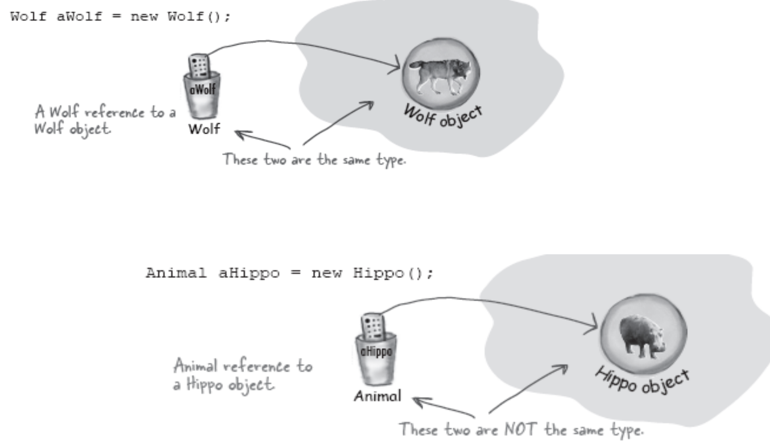
19

Did we forget something?



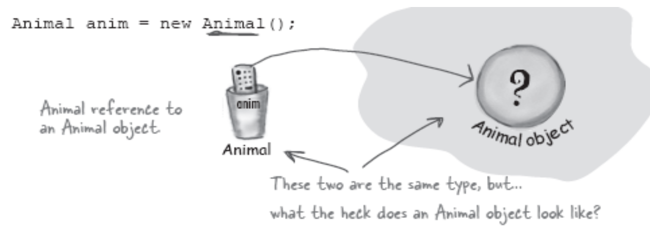
20

These both work….



21

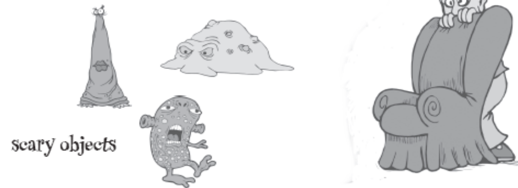
But what about this?



22

??

What does a new Animal() object *look* like?



What are the instance variable values?

Some classes just should *not* be instantiated!

23

Abstract Classes

- We need Animal to provide the inheritance
- But programmers should only instantiate the subclasses where the instance variables have meaning
- Use the keyword Abstract
  - Compiler won't let an abstract class be instantiated

```
abstract class Canine extends Animal
{
public void roam() { }
}
```

24

Abstract Class

- A class that has at least one abstract method is called an *abstract class*
  - An abstract class must have the modifier **abstract** included in its class heading:

```
public abstract class Employee
{
  private instanceVariables;
  . . .
  public abstract double getPay();
  . . .
}
```

25

Abstract Methods

- Methods can also be marked abstract
- Abstract methods have no body
  - public abstract void eat();
- A class with even one abstract method must also be declared abstract

26

Abstract Class

– An abstract class can have any number of abstract and/or fully defined methods

– If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add **abstract** to its modifier

- A class that has no abstract methods is called a *concrete class*

27

Using an Abstract reference

```
public class MakeCanine {
   public void go() {
      Canine c;
      c = new Dog();
      c = new Canine();
      c.roam();
   }
}
```

This is OK, because you can always assign a subclass object to a superclass reference, even if the superclass is abstract.

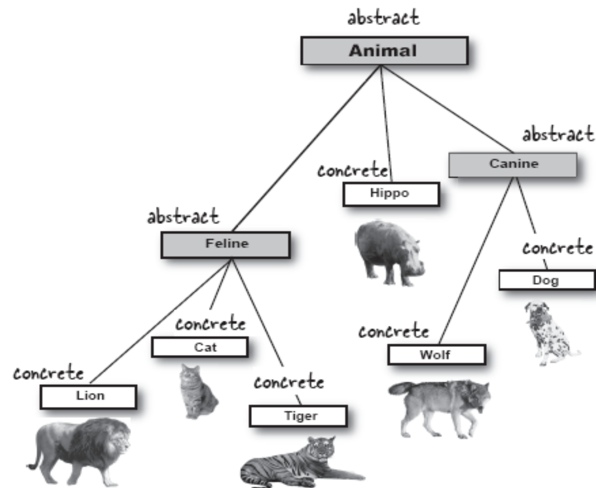
class Canine is marked abstract, so the compiler will NOT let you do this.

File Edit Window Help BeamMeUp

```
% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
       c = new Canine();
           ^
1 error
```

28

Which classes should be abstract?



29

Why?

Q: **What is the *point* of an abstract method? Isn't the whole point of an abstract class was to have common code that could be inherited by subclasses.**

- Inheritable method implementations are **A Good Thing** in a superclass w*hen it makes sense.*
- often *doesn't* make sense, because you can't come up with any generic code that subclasses would find useful.
  - Using an abstract method is that even though you haven't put in any actual method code, you've still defined part of the *protocol* for a group of subtypes (subclasses).

Q: **Which is good because...**

- Polymorphism! Remember, what you want is the ability to use a superclass type (often abstract) as a method argument, return type, or array type.
- That way, you get to add new subtypes (like a new Animal subclass) to your program without having to rewrite (or add) new methods to deal with those new types.
- Remember the Vet class?
  - if it didn't use Animal as its argument type for methods. You'd have to have a separate method for every single Animal subclass!
    - Lion, Wolf, ... you get the idea.
- By using an abstract method, you're saying, "All subtypes of this type have THIS method." for the benefit of polymorphism.

30

Deriving from an Abstract Class

I have wonderful news, mother. Joe finally implemented all his abstract methods! Now everything is working just the way we planned...

Implementing **an abstract method is just like** ***overriding*** **a method.**

- You must implement ALL abstract methods

31

Using Polymorphism

```
public class MyDogList {
    private Dog [] dogs = new Dog[5];     // Use a plain old Dog array behind the scenes.
    private int nextIndex = 0;            // We'll increment this each time a new Dog is added.

    public void add(Dog d) {
        if (nextIndex < dogs.length) {    // If we're not already at the limit
            dogs[nextIndex] = d;          // of the dogs array, add the Dog and print a message.
            System.out.println("Dog added at " + nextIndex);
            nextIndex++;                  // increment, to give us the next index to use
        }
    }
}
```

Now I need to have cats in my list…
what happens?

32

Using Polymorphism

Building our own Animal-specific list

```
public class MyAnimalList {

    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
       if (nextIndex < animals.length) {
          animals[nextIndex] = a;
          System.out.println("Animal added at " + nextIndex);
          nextIndex++;

       }
    }
}
```

Don't panic. We're not making a new Animal object; we're making a new array object, of type Animal. (Remember, you cannot make a new instance of an abstract type, but you CAN make an array object declared to HOLD that type.)

version 2

MyAnimalList
Animal[] animals int nextIndex
add(Animal a)

- Animal objects understand Animal messages
- A Dog object as an Animal will respond in a Dog way (woof) to Animal messages
- A Cat object as an Animal will respond in a Cat way (meow) to Animal messages
- An Animal object held in an Animal variable, or taken from an Animal array will respond to only Animal messages
  - Dog specific messages will be rejected even if instance stored is a Dog
  - However can cast the Animal variable to type Dog
    (Dog) animals[5]

33

Java Interfaces

- Similar to abstract classes
- No implementation components to it (all abstract)
  - Just contains
    - an interface name
    - a list of methods (messages)
- Name corresponds to a polymorphic Type
- All methods listed in interface must be detailed in the "implementing" class

34

Java Interfaces

- Keyword "implements"

public class Dog extends Animal implements Ownable, Movable { }
public class OfficeChair extends Chair implements Ownable, Movable { }

- notice that:
  - both a Dog and OfficeChair instance are movable and own-able
    - even though OfficeChairs are not animals and cannot inherit movable methods from them
    - even though not all animals are own-able (except maybe by zoos)
  - chairs are not moveable in general (see many UofG classroom chairs)
    - shouldn't implement moveable methods in Chair class to be inheritted
  - while you can only inherit from one class, you can implement many different interfaces, as both Dog and OfficeChair do
- Java interfaces is Java's implementation of OO design's Types and sub-Types

35

Back to the regularly scheduled lecture…

Interfaces / Implementation Types Continued

- Objects can handle the messages in any number of different ways
  - through running local methods
  - through running inherited methods
  - by delegating to a collaborating object (i.e. passing the buck)
    - *very important*
- How do you find what messages an object should accept, and which objects it can collaborate with (pass the message off to if it can)?
  - CRC Workshops

37

When Translated to the Technical Model

- External/Visible (public) Methods
  - this is the implementation of the objects interface
- Interior/Hidden (non-public) methods
  - this is the implementation of all self messages
- Instance Variables
  - interior state of the object
  - changes to these values often cause state changes
  - never seen directly from the outside

38

Don't Use Inside-Out Design

- Common mistake
- e.g.
  - "need ***Informant*** class for Poirot system"
  - "so what data should I store?"
    - "how about *name*, *usualMeetingPlace*, and the *handler*"
  - "Oh, now I need get and set methods for them"
    - getName(), setName(),
getUsualMettingPlace(), setUsualMeetingPlace()
getHandler(), setHandler()
  - "Wow, I have a class and look how much I have already done!"
  - "What a nice class…I wonder who will be messaging it?"

39

CRC

“Class” Responsibility Collaboration

- Origins
  - Yet another excellent invention of Cunningham and Beck; at yet another OOPSLA
  - Interesting, among many other reasons, because it wasn’t based on some older, legacy technique
- Still neglected and under-promoted
  - Because there is no UML diagram called a CRC diagram?

40

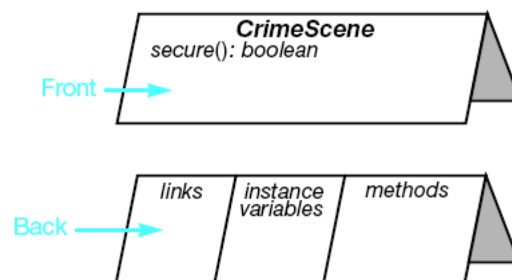
CRC Benefits

- Better objects
  - Outside-in
  - Client and message centric
- “Anthropomorphic” thinking
  - There won’t be a program; so there’s no point in thinking like a programmer
- Team benefits
  - Improved understanding of object technology
  - Improved understanding of application
  - Team gelling
- Fun

41

CRC Practice

- CRC cards
  - Tent cards perhaps
  - Interface message signatures on the visible front; implementation notes on the hidden back



42

… CRC Practice

- Getting started
  - Prioritized use cases can drive the CRC workshops
  - One CRC per use case
    - (and neither the CRC nor the use case should have branches)
  - The use case trigger event allows us to think about the first message of the sequence, into the system
  - The use case outputs and state changes (i.e. stakeholder/actor goals) give goals to the CRC session
  - A good idea is to posit a reasonable interface, e.g. GUI,
    - do not get hung up on designing the interface,
    - let your mental picture of the interface help you to formulate the nature of the first message to an application object

… CRC Practice

- The moderator
  - Runs the workshop timetable
  - Watches over meeting psychology/sociology
  - Ensures good record keeping
  - Ensure necessary follow-up followed up
    - who collects cards
    - who is responsible for interaction diagrams (and when)

… CRC Practice

- The moderator
  - Make sure everyone follows the workshop rules, e.g.
    - Play an instance; you can't play a class
    - Ensure that messages are clear and succinct
      - they should be very close to the actual signature to be used in code
    - Don't get too involved in instantiation/constructors
    - Don't flow chart
      - if a branch is hit, decide which one lies along "main success path"
      - alternate path left for another CRC session
    - Note new kinds of objects and emergent changes to the static model
    - Make sure players actually fill in the cards

45

… CRC Practice

- The moderator
  - Enforces, Software Engineering rules e.g.
    - Respect the (insightful version of) the Law of Demeter
    - High cohesion; low coupling (e.g. one argument or none)
    - Model/View separation
    - Avoid "processors, "handlers" and "managers"
  - Enforces good object technology rules
    - State is secret
      - no one should look at the back of another's card or ask for that information
    - Avoid the "God" object
      - a player may take on too much responsibility (should turn down more messages)
      - Process(or), handle(r) and manage(r) objects in play (responsibilities too vague)
      - High-level objects can exist, such as the restaurant object (make sure that they don't take on too much responsibility)

46

… CRC Practice

- The players
  - A player plays an *instance* (neither when designing types nor when designing classes)
    - Card is the type or class, not the player
    - It can be (usually is) an anonymous instance
  - A player is asked if (s)he'll take responsibility for the message under consideration
    - For all but the most hopelessly inept players, the player is the final arbiter
  - Should I take responsibility; could I take responsibility?
  - Sometimes easier to be negative – why wouldn't I take the responsibility?
    - Not in my character
    - Not enough (or too much) input (message arguments and coupling)
    - Not clear on result (cohesion and/or return problem)
    - Identity crisis – couldn't sensibly be any particular instance
    - Existential crisis – where did I come from?
    - Visibility crisis – message sender and I are not in a position to communicate

47

… CRC Practice

- Collaboration
  - Too big a job is not a reason for refusal
    - Because one can, one is expected to, delegate
    - (As long as it is *one job* (cohesion))
  - It's extremely important to explore all collaboration
    - The majority of messages an instance receives come from other instances
  - When an instance accepts responsibility, we don't cheer and go home,
    - Player forms an opinion as to whether the job could be done in 7 ± 2 lines of executive code,
    - And if it couldn't player looks for another instance (another player) to message to help out with one or more sub-jobs
    - And then that player decides whether to take responsibility and whether to delegate, and …

48

CRC Session in Action - Set-Up

- Everyone given material
  - Use case(s) under consideration
  - Entity-Relation diagram
  - CRC card
- Group initially decide which entities will be needed for completion of the use case
  - each player will play an identified entity / object
- Use case read out by the moderator
- The moderator passes control over to the object/entity messaged by the external actor (or timed event)

49

CRC Session in Action - Regular Play

- If the player accepts the message for the object being messaged
  - writes down message on the front of the card
  - writes down any data to be stored under instance variables on back of the card
  - quickly determines what methods might be need to respond to message
    - writes these methods down on back of the card

50

CRC Session in Action - Regular Play

- During the method determination phase,
  - the player decides if the object can process the message alone (or with information gathered so far)
  - if yes
    - return result,
    - control passed back to moderator/calling object
  - if no, try to find some object to help
    - send message to the object
    - if other object/player rejects message:
      1. modify message and try again, or
      2. find another object to send message to

51

CRC Session in Action - New Entities

- If no one accepts a message then a new entity may be needed
- A message is sent to a delegate object's role-player (not the moderator)
  - delegate object's player will write out a new card for the new object and ask another player (who isn't yet playing an object) to take on the new object
- New entities may also be needed to pass as arguments
- The moderator should note any new entities that arise
  - if possible they should be retrofitted into the Analysis model

52

CRC Session in Action - Messages Sequence

- Sequence of message passing
  - very important
  - provides impute into sequence diagrams
  - One person should be assigned to record the sequence
    - should have no other duties if possible, but probably will be a player in small groups

53

Keep in Mind: The Law of Demeter

- Demeter the Greek goddess of the harvest
  - name used in an early object adoption project
- Laws:
  - If you delegate, delegate fully
  - Don't message your delegate object's objects
    - Don't say "Give me your arrest record, I want to add an item"
    - Rather say "You know your arrests, here's another one for you to remember"
  - Don't be aware of how some object works
    - Don't work at a lower level than is necessary
    - e.g. don't get details of two Matrices and do multiplication of them, send a matrix another matrix in a message and let it do the multiplication

54

CRC Cautions

- Don't let them wander; make progress
- Don't let them drag on; keep to a absolute maximum of 2 hours
- Don't confuse use cases and use case diagrams with CRC and sequence diagrams
- Have the right granularity and the right number of people
  - Too big/too few: players will have to play more than one kind of instance – almost impossible
  - Too small/too many: people will start to feel they're wasting their time
- Don't have a suggester play their suggestion (or a class' designer their class)
- Don't lose the cards or the sequence notes

55

CRC Outputs

- Object type interfaces
  - The card fronts
- Messages and message sequences
  - Who records the notes on message detail and sequence?
- Implementation ideas
  - The card backs
  - Relationship directionality
- Intangibles
  - Team gelling
  - Indoctrination
    - Application
    - Object technology

56

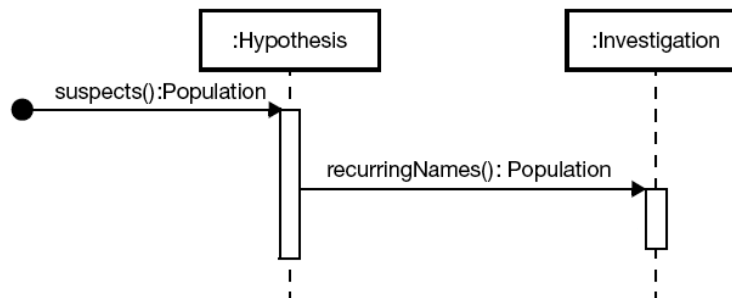
Interaction Diagrams

(Or sequence diagrams)
- These are a most important additional companion to the static structural (“class”) diagrams early OO design focused on
- They are illustrative
  - One could never diagram every possible message sequence
- They involve instances rather than classes
- They have (had) two forms
  - One focusing on time-ordering of messages
    - Sequence Diagrams
  - One focusing on the structural context of the instances
    - Collaboration Diagrams (not used much in UML 2.0)
- They involve messages
- Unlike static (“class”) diagrams,
  - They have less historical baggage
  - It’s less easy to get confused and start designing databases
  - Especially if developed through CRC, they are outside-in rather than naïve/inside-out

57

… Interaction Diagrams

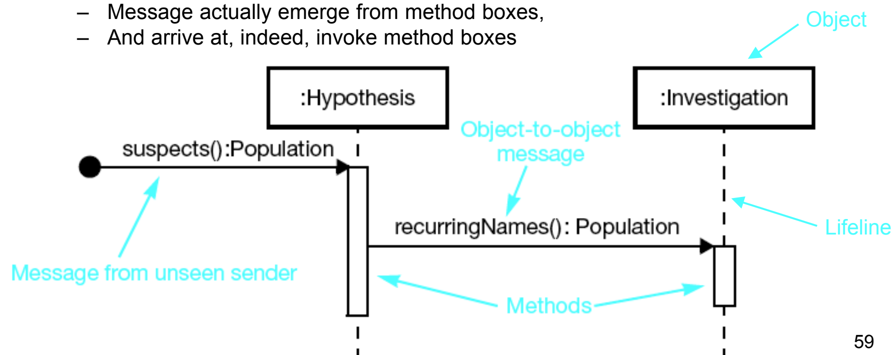
- The unique feature is an instance lifeline for each instance
  - A vertical line (the rails)
- Messages are directed from instance to instance
  - Horizontal lines (the rungs)
- The remaining essential element is the method box (an “activation” in the UML)
  - Message actually emerge from method boxes,
  - And arrive at, indeed, invoke method boxes



58

… Interaction Diagrams

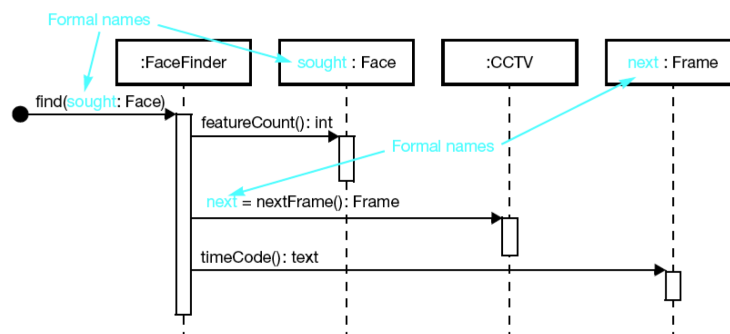
- The unique feature is an instance lifeline for each instance
  - A vertical line (the rails)
- Messages are directed from instance to instance
  - Horizontal lines (the rungs)
- The remaining essential element is the method box (an "activation" in the UML)
  - Message actually emerge from method boxes,
  - And arrive at, indeed, invoke method boxes



59

Sequence Diagrams: Types & Formal Names

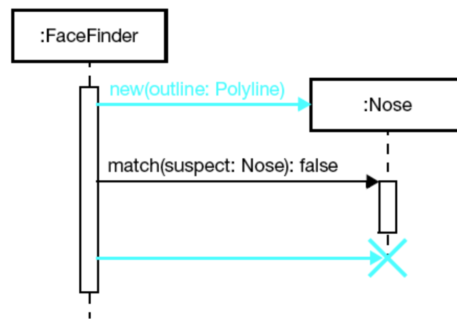
- Objects (boxes at the top of the diagram)do not have names
  - Objects not classes, nor are they variables
  - Do have a type (after the colon)
- May need to distinguish different objects of the same type
  - then can give a name in front of the colon
    - make it **meaningful** in terms of the **design**



60

Sequence Diagrams: Creation & Deletion

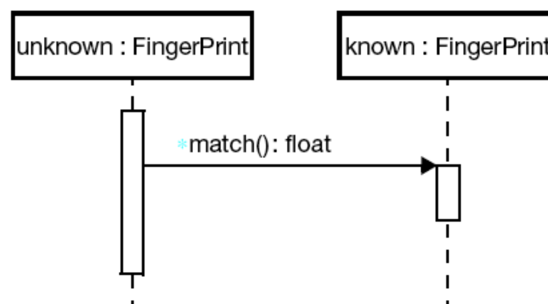
- A message can result in the creation of an object as well as the invocation of a method
- An object can also be destroyed
  - not import for Java, which has a Garbage collector
  - very important for C++



61

Sequence Diagrams: Iteration

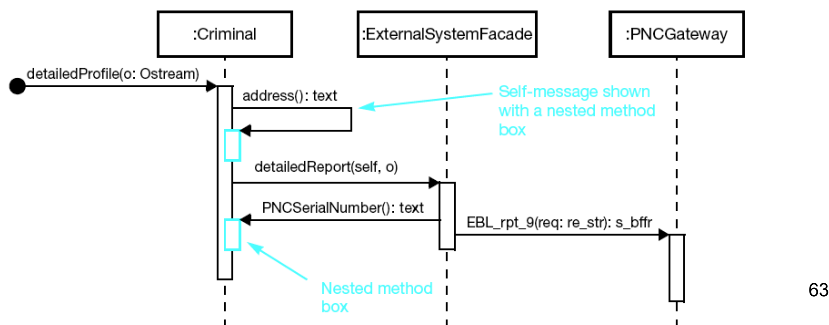
- Can we represent a message that will be sent many times (in a loop)
  - yes … put a * in front of it



62

Sequence Diagrams:Self-Messaging & Nested methods

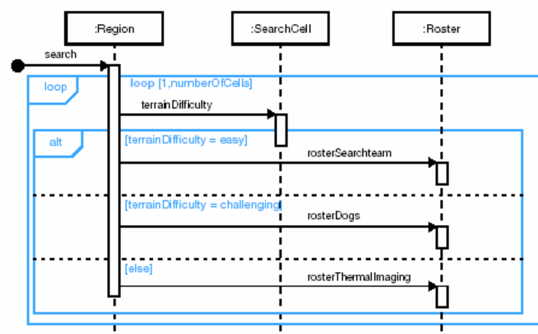
- An object (often) can send a message to itself
  - called self-messaging
  - also called recursion in UML 1.0, but it is not actually recursive
- Causes a "nested" method to be invoked
  - small method box overlapping larger method box



63

Sequence Diagrams: Decisions and Branching

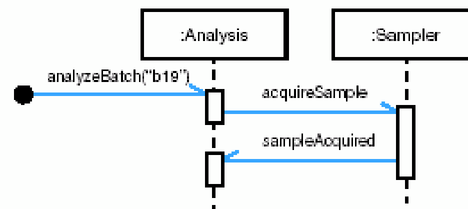
- Decisions and Branching
  - still cumbersome diagramatically
  - in OO code lots of iteration, little branching
  - becomes more like a flowchart
    - too low level



64

Sequence Diagrams: Asynchronous Messages

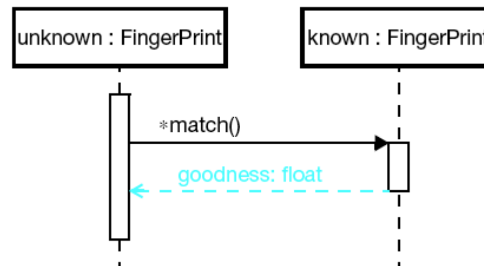
- One-way communication
- sender doesn't block
- no reply
  - if there is a reply it will be another asynchronous message in the opposite direction
- used with CORBA or RMI



65

Sequence Diagrams: Return Arrows

- Shows when a return from a method happens
  - redundant
    - always return control to the caller
    - implied by the bottom of the method box
  - clutters diagram
  - don't use



66

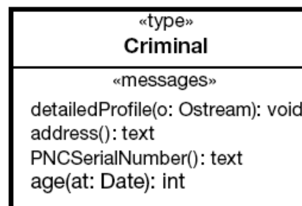
Structure Diagrams Revisited

- Slight modifications from Subject Model structure diagram (entity-relationship diagram)
  - corrections and extension
    - if CRC workshop showed us a new entity or were wrong about one, we must correct it
  - structural extracts to accompany and complement the sequence diagrams
  - message signature detailing
  - navigability arrow stabilizing

67

…Structure Diagrams Revisited

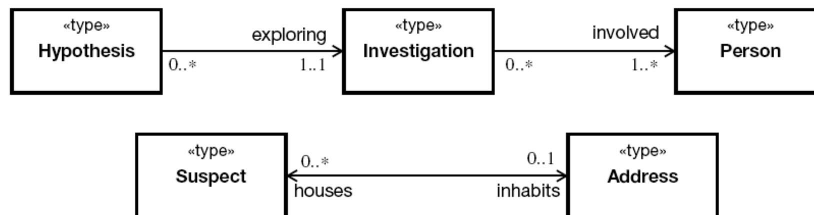
- Depicting object types
  - Object types are depicted with a box, as usual
  - But with a `«type»` keyword ("stereotype")
  - They are characterized only by the messages they accept; so there's still one substantive compartment specifying those messages
  - message signatures more complete, possibly with meaningful argument names
  - Object Types can now be used for message return, not just primitives (still rare)



68

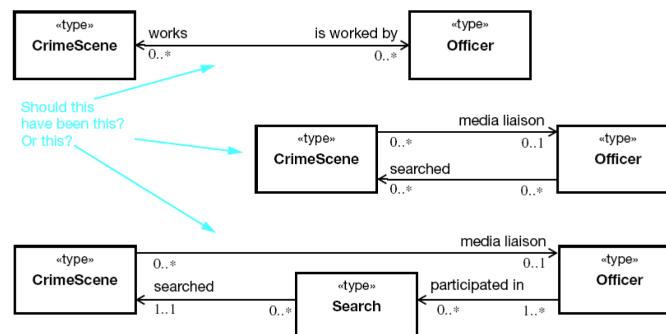
… Structure Diagrams Revisited

- Depicting direction in characterizing relationships
  - With evidence building up from the CRCs,
  - The non-existence-dependent, characterizing relationships ("associations") can be given their definitive directions
  - We show direction with an arrowhead at one end, the other end or both ends
  - The analysis suspicions as to direction are now confirmed (or challenged) and arrowheads should accompany role names



… Structure Diagrams Revisited

- A few navigability arrows will be bi-directional (and tricky to implement)
- But with sufficient object types and relationships, most should be directed
  - If lots are bi-directional, assess them again
  - Re-apply the test mentioned in the Analysis chapter



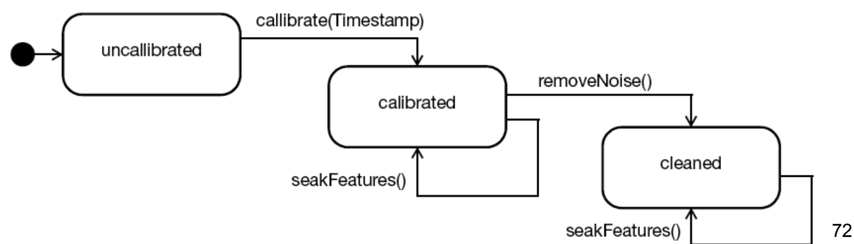
State Machines Revisited

- Protocol state machines
  - State machines can help; but it is very easy to get them wrong
  - One way to tame the beasts is to avoid outputs and allow the message to imply the method
    - Entirely natural for most OO languages of course
  - Such a state machine is type of **protocol state machine**
    - as opposed to a behavioral state machine
    - should be the state machine used while detailing our object types
  - If a state machine is deemed necessary to an object type
    - It's specifying permitted orderings of the messages

71

… State Machines Revisited

- Events and messages
  - Events in the analysis model were "significant happenings"
    - And we didn't ask where they "came from"
  - Events in the design model are messages
    - And they come from other objects or from interfaces



72

…State Machines Revisited

- Message ordering with regard to what or whom?
  - Must be ordered this way in general?
  - Or just with respect to a specific client "using" the object
  - Or represents the order of messages on one thread?
  - Probably Client ordering, but should state explicitly.
- Note: State Machines restricts a Type
  - consequently, less easy to use
  - so use state machines only for a very good reason

73

Second Model Summary

- Object instance specifications (object types)
- Messages
- Sequence diagrams
- (Directions
  - This is parenthetical because, strictly speaking, it's *still* just suspicion
  - Object types don't specify implementation
  - Characterizing relationships are implementation
  - We need an idea as to the awareness that instances will have of other instances
    - Otherwise we can't do the "collaboration" of CRC
  - But strictly speaking it's only an informal notion
  - And it's about to become formalized in the next model)

74