An Example OO Analysis Method

CRC (Class, Responsibility, Collaboration): It clearly identifies the Classes, what the Responsibilities are of each class, and how the classes Collaborate (interact).

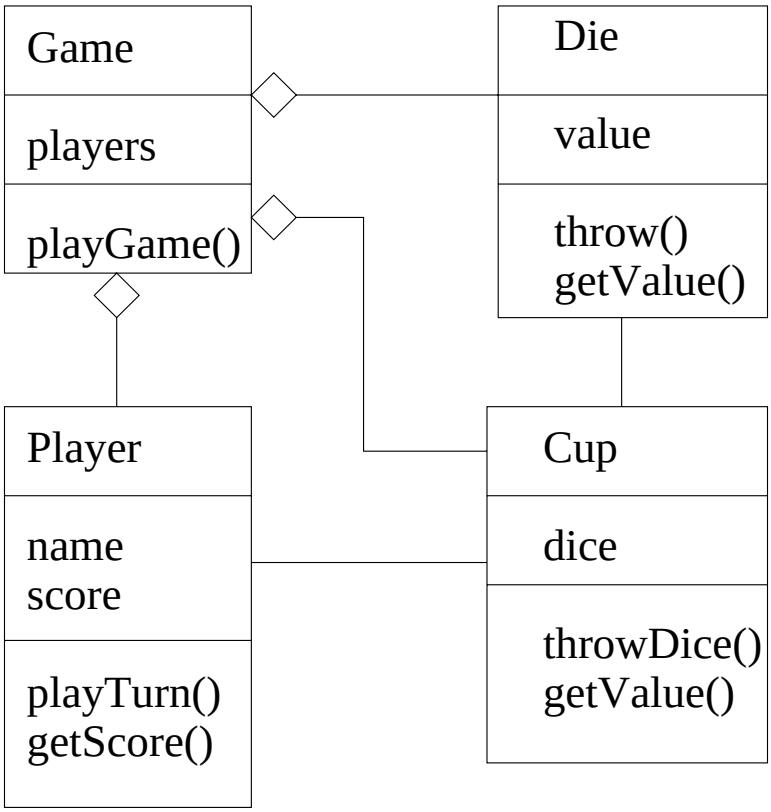
In the CRC method, you draw **class diagrams**:

- each class is indicated by a rectangle containing
  - name of class
  - list of attributes (instance variables)
  - list of methods
- if class 1 is a subclass of class 2, then draw an arrow from class 1 rectangle to class 2 rectangle
- if an object of class 1 is part of (an instance variable of) class 2, then draw a line from class 1 rectangle to class 2 rectangle with a diamond at end.
- if objects of class 1 need to communicate with objects of class 2, then draw a plain line connecting the two rectangles.

The arrows and lines can be annotated to indicate the number of objects involved, the role they play, etc.

CRC Example

To model a game with several players who take turns throwing a cup containing dice, in which some scoring system is used to determine the best score:



This is a diagram of the *classes*, not the *objects*. Object diagrams are trickier since objects come and go dynamically during execution.

Double-check that the class diagram is consistent with requirements scenarios.

Object-Oriented Analysis and Design (cont'd)

While fleshing out the design, after identifying *what* the different methods of the classes should be, figure out *how* the methods will work.

This means deciding what algorithms and associated data structures to use.

Do not fall in love with one particular solution (such as the first one that occurs to you). Generate as many different possible solutions as is practical, and then try to identify the best one.

Do not commit to a particular solution too early in the process. Concentrate on *what* should be done, not *how*, until as late as possible. The use of ADTs assists in this aspect.

Verification and Correctness Proofs

Part of the design includes deciding on (or coming up with new) algorithms to use.

You should have some convincing argument as to why these algorithms are *correct*.

In many cases, it will be obvious:

- trivial action, such as a table lookup
- using a well known algorithm, such as heap sort

But sometimes you might be coming up with your own algorithm, or combining known algorithms in new ways.

In these cases, it’s important to check what you are doing!

Verification and Correctness Proofs (cont'd)

The Standish book describes one particular way to prove correctness of small programs, or program fragments. The important lessons are:

- It is possible to do very careful proofs of correctness of programs.
- Formalisms can help you to organize your thoughts.
- Spending a lot of time thinking about your program, no matter what formalism, will pay benefits.
- These approaches are impossible to do by hand for very large programs.

For large programs, there are research efforts aimed at *automatic program verification*, i.e., programs that take as input other programs and check whether they meet their specifications.

Generally automatic verification is slow and cumbersome, and requires some specialized skills.

Verification and Correctness Proofs (cont'd)

An alternative approach to program verification is proving algorithm correctness.

Instead of trying to verify actual code, *prove the correctness of the underlying algorithm*.

- Represent the algorithm in some convenient pseudocode.
- then argue about what the algorithm does at higher level of abstraction than detailed code in a particular programming language.

Of course, you might make a mistake when translating your pseudocode into Java, but the proving will be much more manageable than the verification.

Implementation

The design is now fleshed out to the level of code:

- Create a Java class for each design class.
- Fix the type of each attribute.
- Determine the signature (type and number of parameters, return type) for each method.
- Fill in the body of each method.

As the code is written, document the key design decisions, implementation choices, and any unobvious aspects of the code.

Software reuse: Use library classes as appropriate (e.g., Stack, Vector, Date, HashTable). Kinds of reuse:

- use as is
- inherit from an existing class
- modify an existing class (if source available)

But sometimes modifications can be more time consuming than starting from scratch.

Testing and Debugging: The Limitations

Testing cannot prove that your program is correct.

It is impossible to test a program on every single input, so you can never be sure that it won't fail on some input.

Even if you could apply some kind of program verification to your program, how do you know the verifier doesn't have a bug in it?

And in fact, how do you know that your requirements correctly captured the customer's intent?

However, testing still serves a worthwhile, pragmatic, purpose.

Test Cases, Plans and Logs

Run the program on various **test cases.** Test cases should cover every line of code in every method, including constructors. More specifically,

- test on valid input, over a wide range
- test on valid inputs at the limits of the range
- test on invalid inputs

Organize your test cases according to a **test plan**: it is a document that describes how you intend to test your program. Purposes:

- make it clear what a program should produce
- ensure that testing is repeatable

Results of running a set of tests is a **test log**: should demonstrate that each test produced the output predicted by the test plan.

After fixing a bug, you must rerun your ENTIRE test plan. (Winder and Roberts calls this the Principle of Maximum Paranoia.)

Kinds of Testing

Unit testing: test a method M all by itself using

- a **driver** program that calls M with the arguments of interest, and
- **stubs** for the methods that M calls – a stub returns some hard coded value without doing any real calculations.

Integration testing: test the methods combined with each other. Two approaches to integration testing:

Bottom-up testing Start with the "bottom level" methods and classes, those that don't call or rely on any others. Test them thoroughly with drivers.

Then progress to the next level up: those methods and classes that only use the bottom level ones already tested. Use a driver to test combinations of the bottom two layers.

Proceed until you are testing the entire program.

Kinds of Testing (cont'd)

Top down testing proceeds in the opposite direction, making extensive use of stubs.
Reasons to do top down testing:

- to allow software development to proceed in parallel with several people working on different components that will then be combined – you don't have to wait until all levels below yours are done before testing it.
- if you have modules that are mutually dependent, e.g., X uses Y, Y uses Z, and Z uses X. You can test the pieces independently.

Other Approaches to Debugging

In addition to testing, another approach to debugging a program is to visually **inspect** the code – just look through it and try to see if you spot errors.

A third approach is called a **code walk through** – a group of people sit together and "simulate", or walk through, the code, seeing if it works the way it should.

Some companies give your (group's) code to another group, whose job is to try to make your code break!