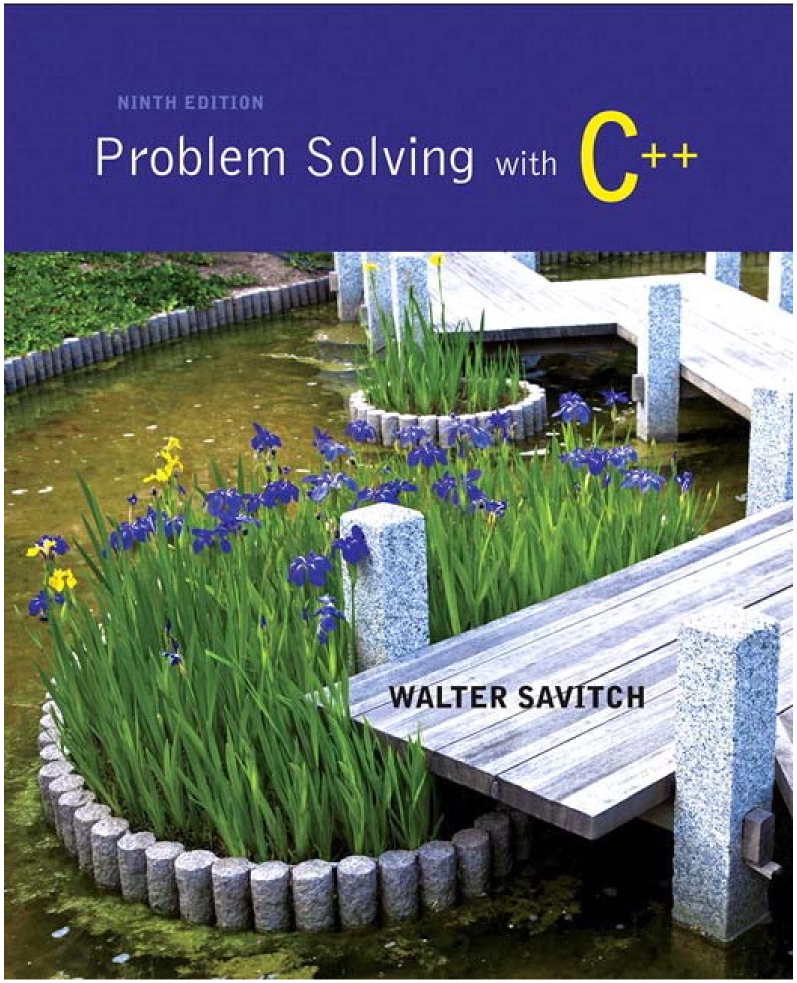
SFU SIMON FRASER UNIVERSITY BURNABY

Designing Classes
Chapter 10

Instructor: Scott Kristjanson

CMPT 135

SFU Surrey, Spring 2016



Scope

Writing your own Classes and Methods:

- Data Flow Diagrams and Structured Analysis
- Identifying classes and objects
- Structure and content of classes
- Member data
- Visibility modifiers
- Method structure
- Constructors
- Relationships among classes
- Friend methods and data

Getting Started

So how does one get started designing a software project?

Do *not* start by writing Class Definitions – Big Mistake!

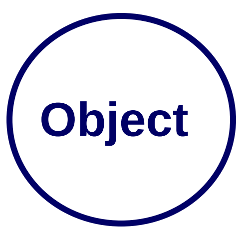
Do *not* start by defining your Data and Attributes – **Bigger Mistake!!**

Start by thinking about the problem to be solved!

- Write down a description of the problem in words
- What are the "Things" involved? What are the Nouns?
- What actions can can happen to these things? What are the Verbs?

At the top level, the Nouns will become your Object Classes.

- Start drawing the top level objects as Circles on paper or a white board
- Do not worry about how to code this, just draw it out!



Understanding the Problem Statement

Read the problem description

```
Problem Description:
 Write a program to simulate what happens when a
 gambler rolls two dice 500 times and then report on
 how many SnakeEyes (double ones) were rolled
```

Before you code it, design it!
Before you design it, understand it!
To understand it, you need to ***start*** with requirements!

What are your Requirements?

- *What problem is being solved?*
- *What does your program need to do?*
- *What specific requirements are specified if any?*

Define your Top-Level Objects first

Read the problem description again and identify the Nouns

```
Problem Description:
 Write a program to simulate what happens when a
 gambler rolls two dice 500 times and then report on
 how many SnakeEyes (double ones) were rolled
```

Identify the Nouns, these are your top level objects

Next draw and label all the objects at a top level

Do not worry about the details, just draw them out.

You are just trying to *understand* the problem, not code it!

Gambler **Dice**

Determine the sets of actions, inputs and outputs

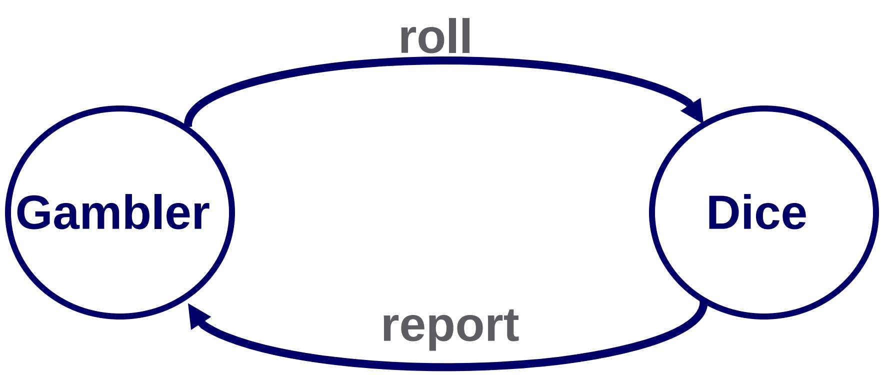
Identify the Verbs in the Problem statement:

```
Write a program to simulate what happens when a gambler
rolls two dice 500 times and then report on how many
SnakeEyes (double ones) were rolled
```

What Actions can happen to your objects?

- Do these Actions require input? - These become parameters
- Do these Actions request output? - These become return values

- These verbs will become your top-level methods
- For Now, draw labeled arrows between your Objects to represent Actions



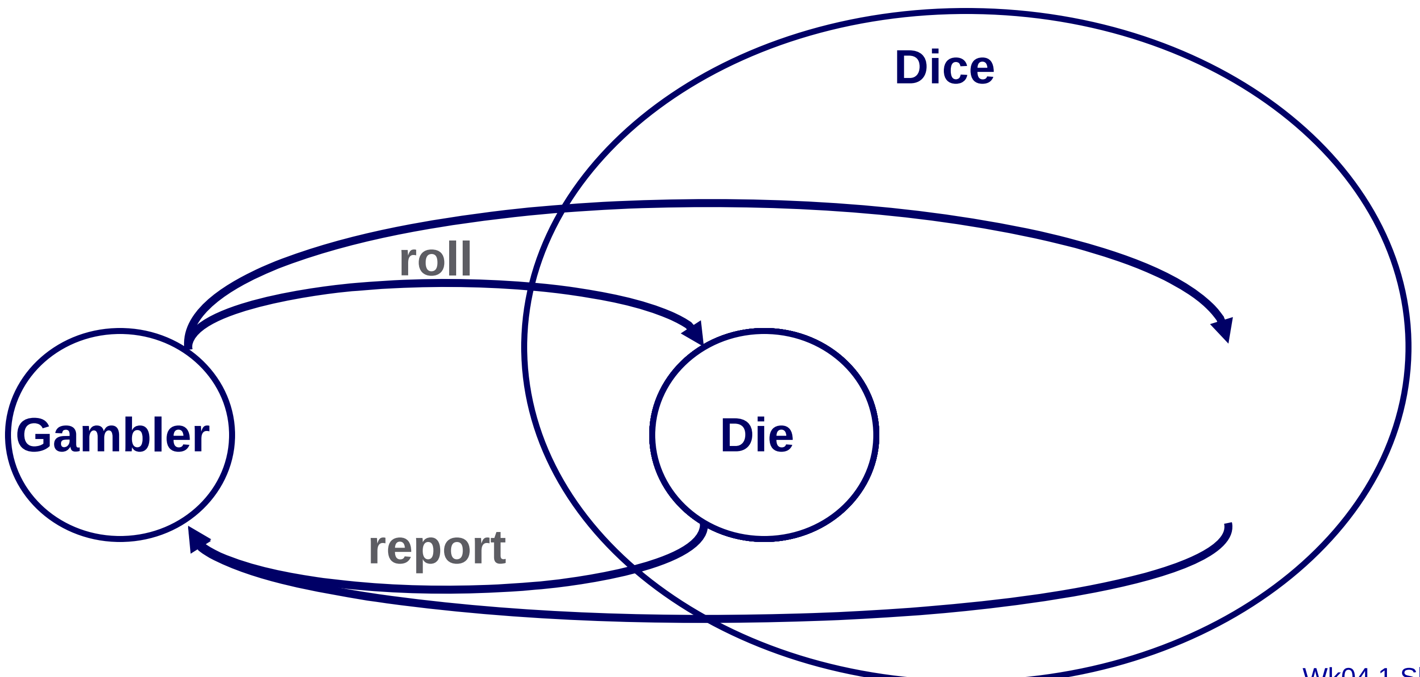
Decompose Complex Objects into Simpler Objects

The Dice object in a diagram represents Two Dice

If that sounds complicated, *decompose* into simpler objects

Let’s decompose Dice into something simpler: a Die.

We will need two instances of Die for this program

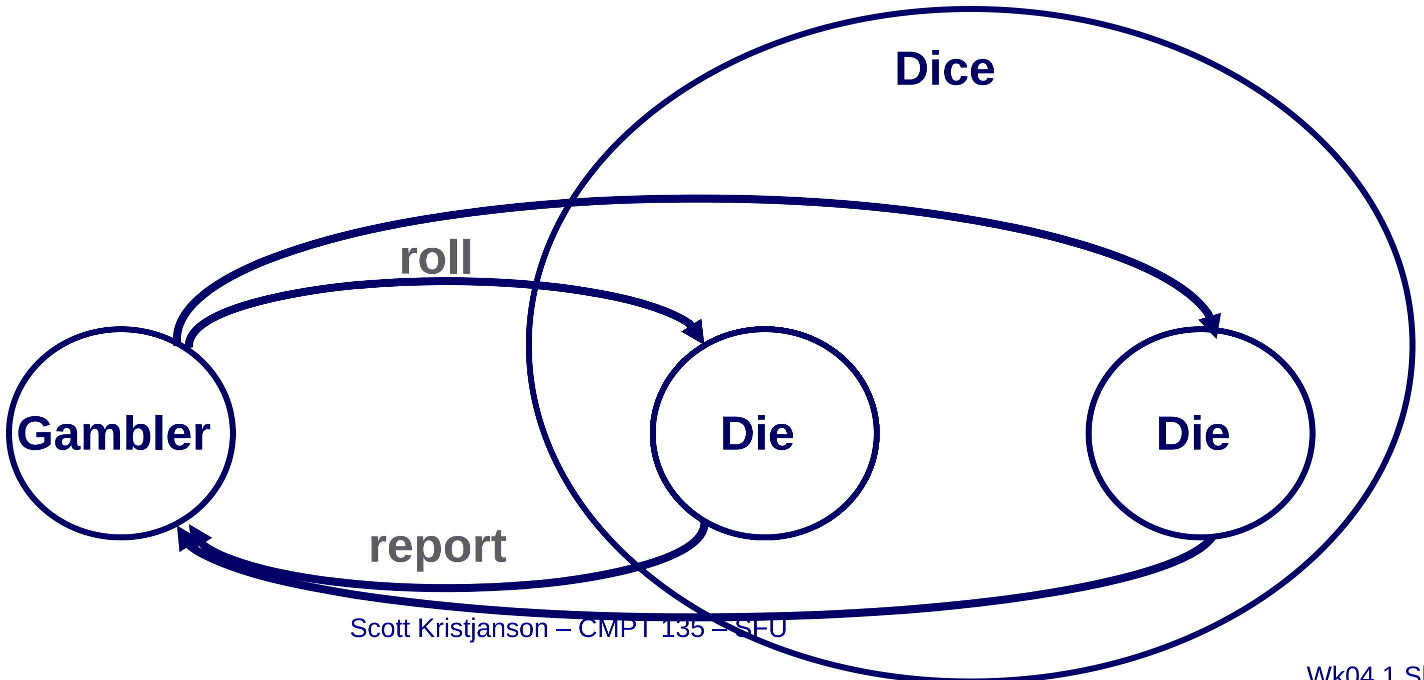


But Wait! We are defining Classes not Objects

Our Goal is to define Classes and Methods, *not* Objects!

The two Die objects are two instances of the same thing.

Map identical objects back into a single class.



A Data Flow Diagram

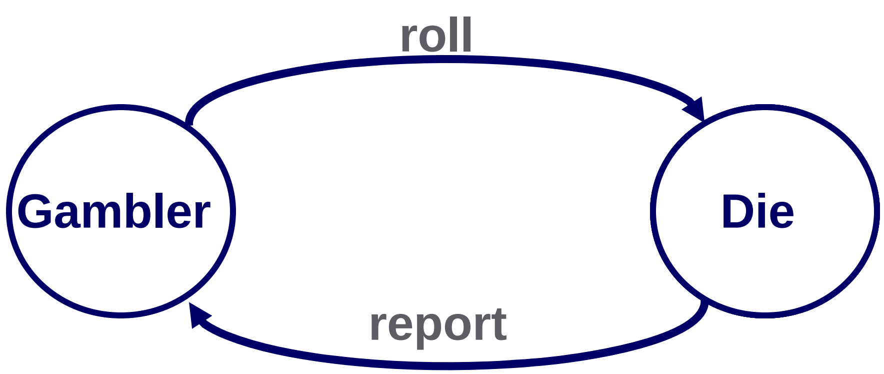
Our completed diagram is called a *Data Flow Diagram*

This design technique is called *Structured Analysis*[3]

Allows for fast design BEFORE you commit it to code

Easier to change a whiteboard diagram than to modify code!

Next… Validate your the Data Flow Diagram



Validating your Data Flow Diagram

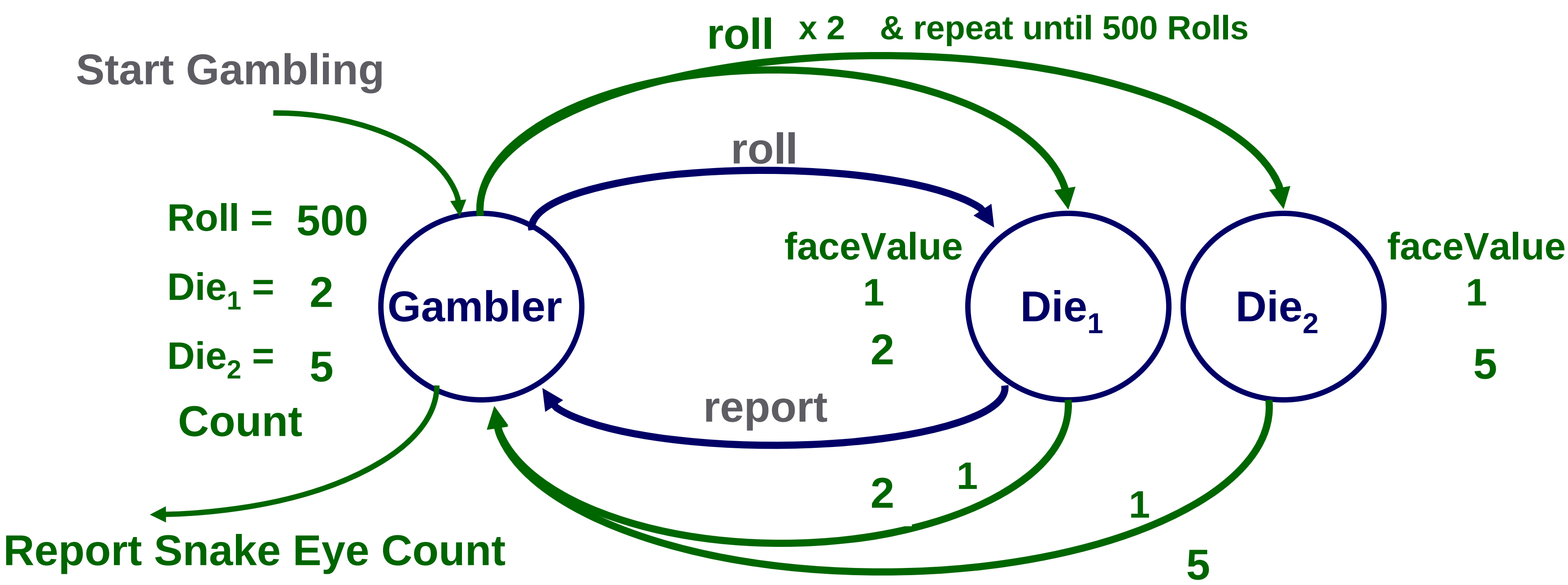
Data flow diagrams show data flow, but not flow of control

A Flow of Control is a specific path through the data flow

Run through a test case's flow of control to test your design

Add State Variables to Objects as needed during your "Run"

Instantiate Objects As Needed

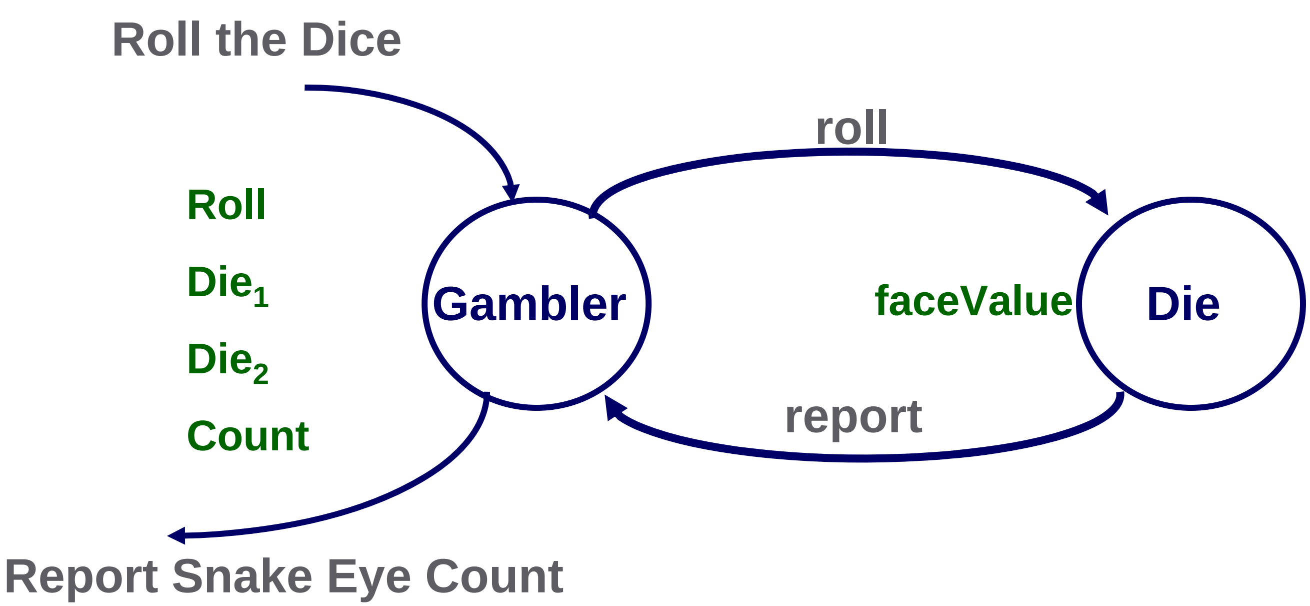


Designing Classes

Create a C++ Class for each Class in the Data Flow Diagram

Create a corresponding method for each arc in the data flow

Create local variables for each state used in your “Run”



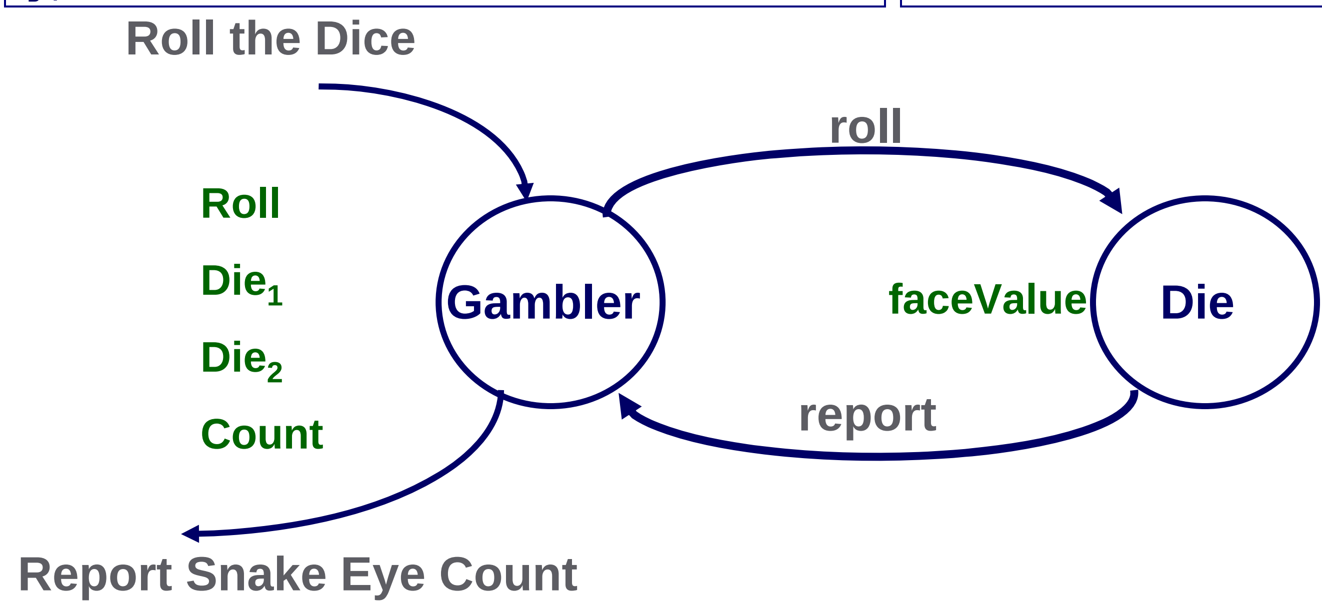
Declaring the Class, Methods, and Members


```
class Gambler
{
 public:
 Gambler(); // Constructor
 int rollTheDice(int MaxRolls);
 /* Returns Count of # SnakeEyes */

 private:
    int roll, count;
    Die die1, die2;
};
```

```
class Die
{
 public:
   Die();     // Constructor
   int roll();
   /* Returns new faceValue */

 private:
   int faceValue;
};
```



Fill in the code for the Methods


```
Gambler::Gambler() // Constuctor
{
 roll  = 0;
 count = 0;
}

int Gambler::rollTheDice(int MaxRolls)
{
 /* Returns Count of # SnakeEyes */
 count = 0;
 for (roll=1;roll<=MaxRolls;roll++)
   if ((die1.roll()+die2.roll())==2)
    count++; // SnakeEyes!

 return count;
}
```

```
Die::Die() // Constructor
 {
  srand(time(0));
  faceValue = (rand()%6)+1;
 }

 int Die::roll()
 {
  // Roll the dice and return faceValue!
  faceValue = (rand()%6)+1;
  return faceValue;
 }
```

And Finally, Create a Main Method and Test

`main` instantiates a **`Gambler`** to test the class

`Gambler` constructor will instantiate two **`die`** objects

```
int main(int argc, char **argv)
{
  const int MAX_ROLLS = 500;
  int       snakeEyes = 0;
  Gambler   gambler;

  snakeEyes = gambler.rollTheDice(MAX_ROLLS);

  cout << "Rolled " << snakeEyes  << " Snake-Eyes in "
       << MAX_ROLLS << "   rolls" << endl;;
}
```

```
Rolled 15 Snake-Eyes in 500 rolls
```

Inter-Object Coupling

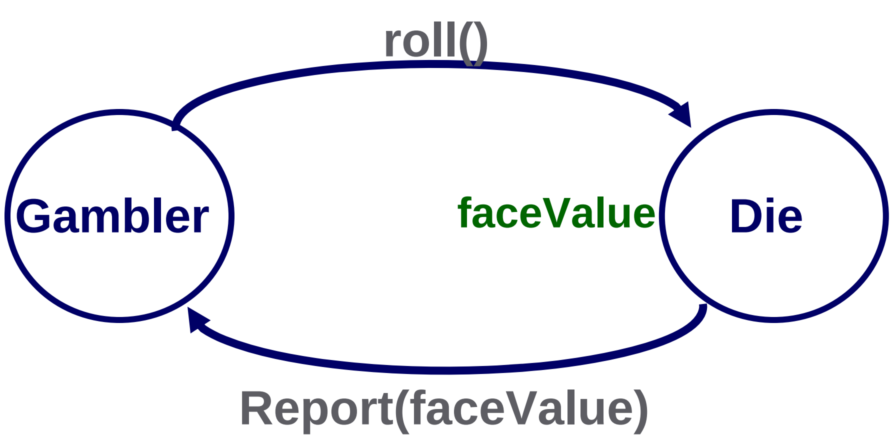
So how do you tell if you have a good design?

Start by looking at Inter-Object *Coupling:* [4][5]

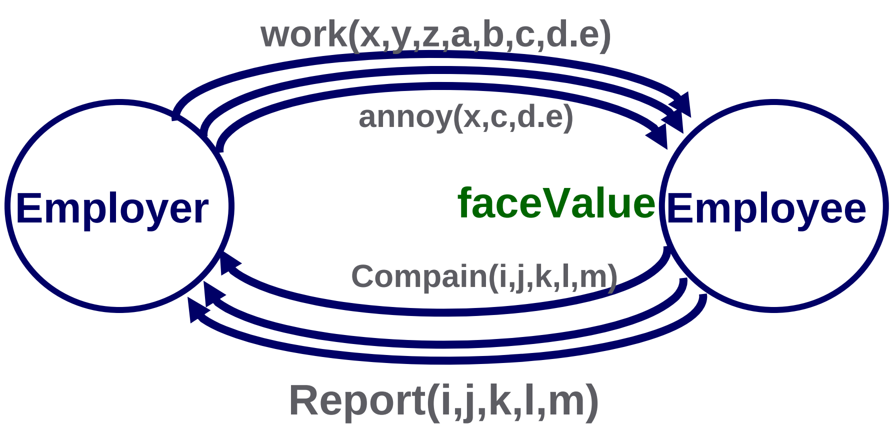
- Complexity of interfaces
- Number of interfaces
- Amount of shared knowlege

Good Design minimizes interface complexity or “thickness”

Low Coupling = Good Design



High Coupling = Bad Design



Object Cohesion

Cohesion is a measure of how connected code is.

A well designed object maximizes Cohesion within an elements of an Object. Good forms of Cohesion include:

- Informational – operates on the same set of internal data objects
- Functional – relate to a single object class or set of functions
- Sequential – methods call other methods as subroutines

Good design minimizes Inter-Object Cohesion

Rules for Good Design

Understand and capture requirements first!

Design before you Code

Top-Down decomposition is essential for solving complex problems

Understanding Dataflow is essential

- Identify all the Nouns and Verbs in your problem statement
- Nouns become Objects
- Verbs become Methods
- Object Data becomes Local Variables
- Data on Arcs become Parameters and Return Values
- *Decompose* complex objects into simpler objects
- If your Dataflow does not fit on one page, *encapsulate* Objects until it does

Minimize:

- Interface Complexity
- Inter-Module Coupling

Maximize:

- Intra-Module Cohesion

Classes and Objects

- An object has state, defined by the values of its attributes
- The attributes are defined by the data associated with the object's class
- An object also has behaviors, defined by the operations associated with it
- Operations are defined by the methods of the class

Classes and Objects

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

Identifying Classes and Objects

A class represents a group (classification) of objects with the same behaviors

Generally, classes that represent objects should be given names that are singular ***nouns***

Examples: **`Coin`**, **`Student`**, **`Message`**

A class represents the concept of one such object

We are free to instantiate as many of each object as needed

Identifying Classes and Objects

One way to find potential objects is by identifying the nouns in a problem description:

```
The user must be allowed to specify each
product by its primary characteristics,
including its name and product number. If the
bar code does not match the product, then an
error should be generated to the message window
and entered into the error log. The summary
report of all transactions must be structured
as specified in section 7.A.
```

Identifying Classes and Objects

Sometimes it is challenging to decide whether something should be represented as a class

For example, should an employee's address be represented as a set of variables or as an `Address` object

The more you examine the problem and its details the more clear these issues become

When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

If set of data is used in multiple places, it should be a struct. If it has multiple methods associated with it, it is a candidate to be a Class.

Identifying Classes and Objects

We want to define classes with the proper amount of detail

For example, it may be unnecessary to create separate classes for each type of appliance in a house

It may be sufficient to define a more general `Appliance` class with appropriate instance data

It all depends on the details of the problem being solved

Identifying Classes and Objects

Part of identifying the classes we need is the process of ***assigning responsibilities*** to each class

Every activity that a program must accomplish must be represented by one or more methods in one or more classes

We generally use ***verbs*** for the names of methods

In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design

Constructors

A *constructor* is a special method that is used to set up an object when it is initially created

A constructor has the same name as the class

The `Die` constructor is used to set the initial face value of each new die object to one

Instance Data

The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it

A class declares the type of the data, but it does not reserve any memory space for it

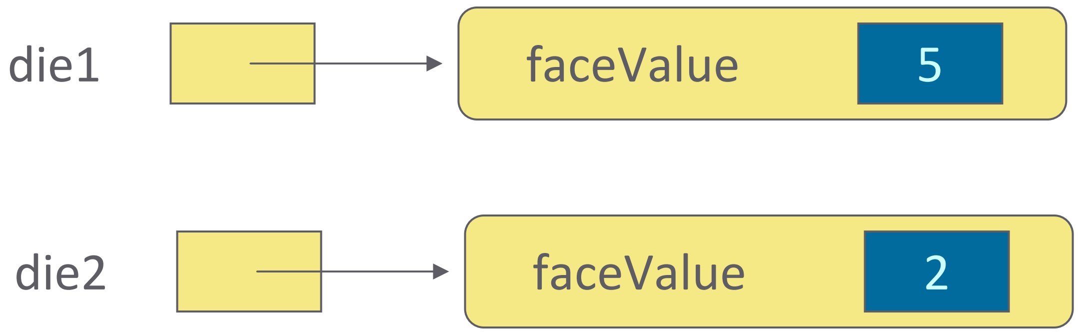
Every time a `Die` object is created, a new `faceValue` variable is created as well

The objects of a class share the method definitions, but each object has its own data space

That's the only way two objects can have different states

Instance Data

We can depict the two `Die` objects from the `SnakeEyes` program as follows:



Each object maintains its own `faceValue` variable, and thus its own state

UML Diagrams

UML stands for the *Unified Modeling Language*

UML diagrams show relationships among classes and objects

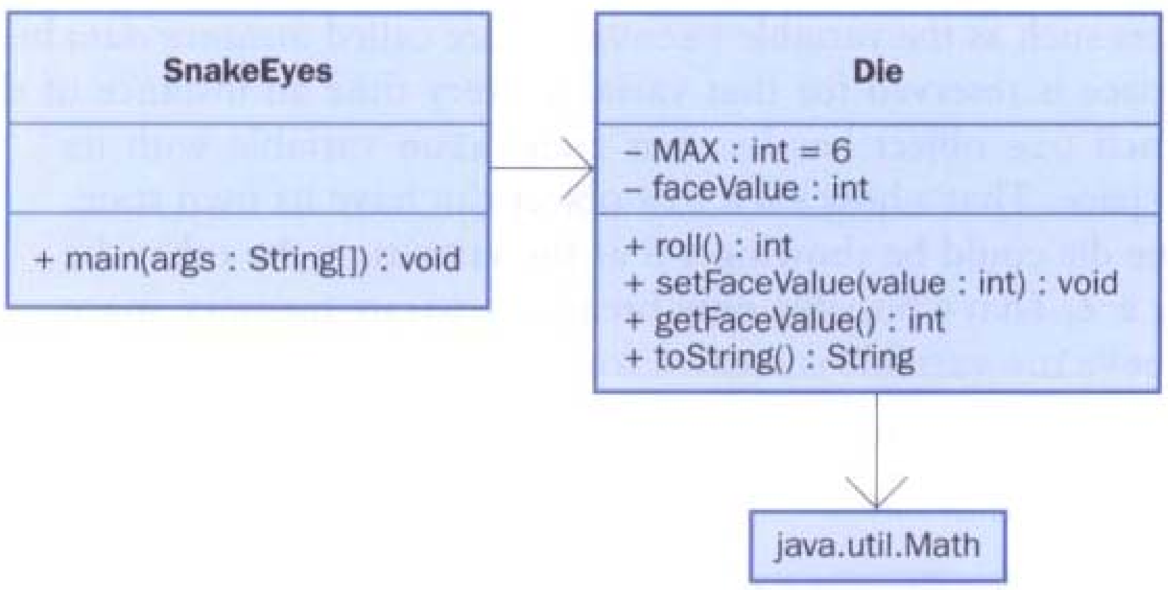
A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)

Lines between classes represent *associations*

A solid arrow shows that one class *uses* the other (calls its methods)

UML Diagrams

A UML class diagram showing the classes involved in the `SnakeEyes` program:



Encapsulation

We can take one of two views of an object

- internal - the details of the variables and methods of the class that defines it
- external - the services that an object provides and how the object interacts with the rest of the system

From the external view, an object is an *encapsulated* entity, providing a set of specific services

These services define the *interface* to the object

Encapsulation

One object (called the *client*) may use another object for the services it provides

The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished

Any changes to an object's state (its variables) should be made by that object's methods

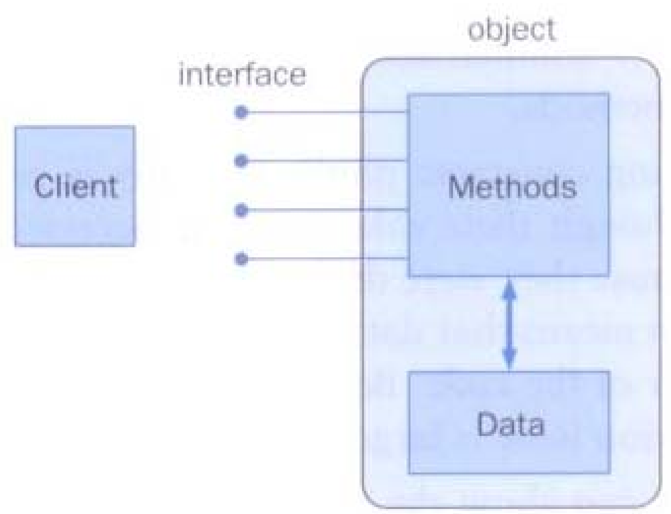
We should make it difficult, if not impossible, for a client to access an object’s variables directly

That is, an object should be *self-governing*

Encapsulation

An encapsulated object can be thought of as a *black box* – its inner workings are hidden from the client

The client invokes the interface methods of the object, which manages the instance data



Visibility Modifiers

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Method Declarations

Let's now examine method declarations in more detail

A *method declaration* specifies the code that will be executed when the method is invoked (called)

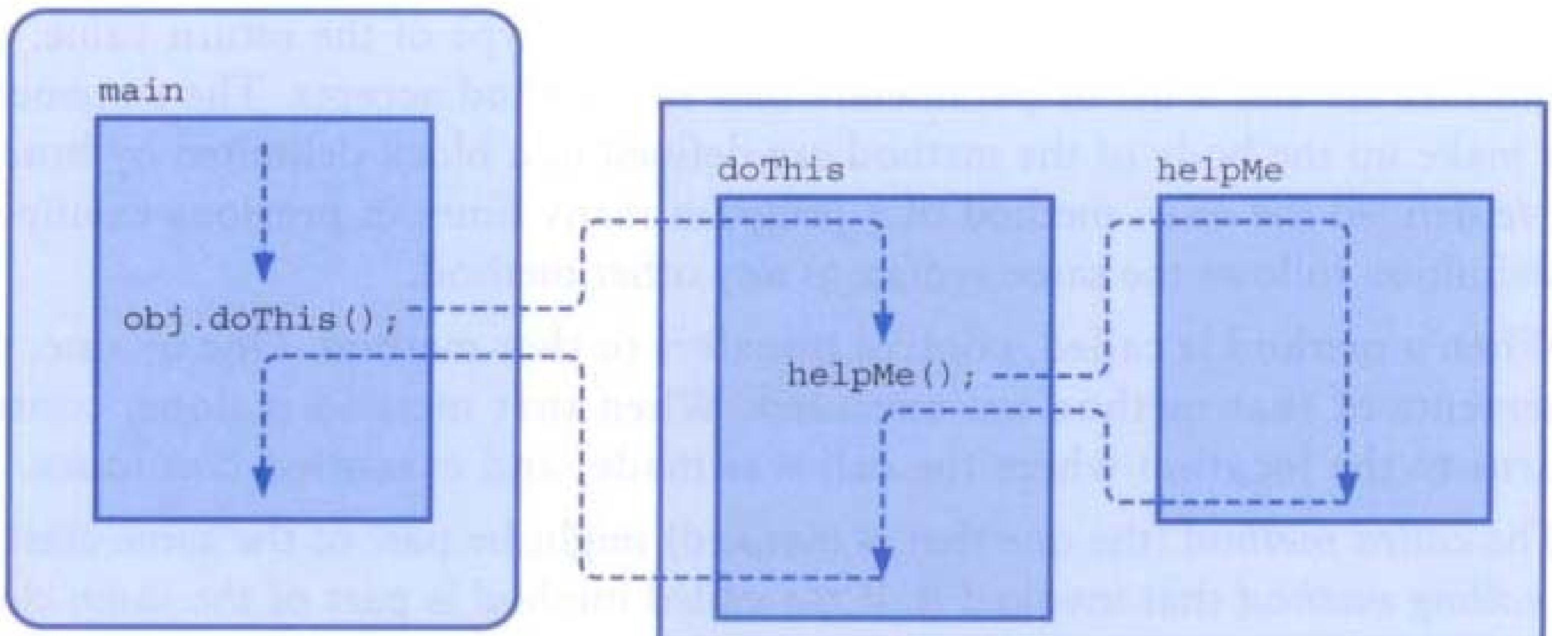
When a method is invoked, the flow of control jumps to the method and executes its code

When complete, the flow returns to the place where the method was called and continues

The invocation may or may not return a value, depending on how the method is defined

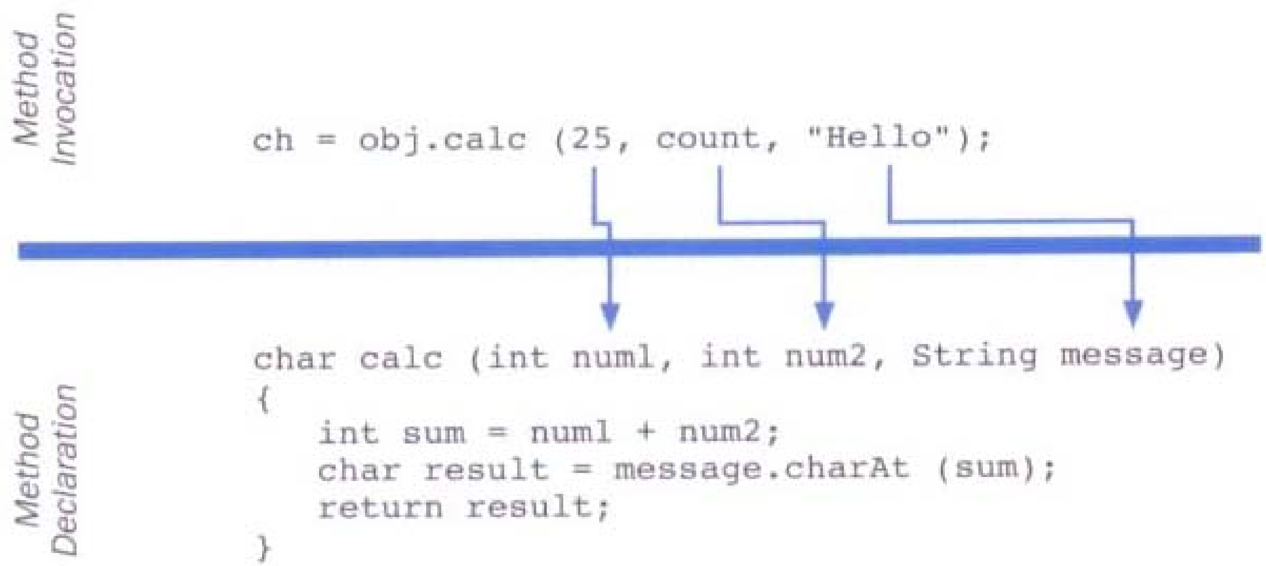
Methods

The flow of control through methods:



Parameters

When a method is called, the *actual parameters* in the invocation are copied into the *formal parameters* in the method header



Class Relationships

Classes in a software system can have various types of relationships to each other

Three of the most common relationships:

- Dependency: A *uses* B
- Aggregation: A *has-a* B
- Inheritance: A *is-a* B

Let's discuss dependency and aggregation further

Dependency

A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other

We've seen dependencies in many previous examples

We don't want numerous or complex dependencies among classes

Nor do we want complex classes that don't depend on others

A good design strikes the right balance

Aggregation

An *aggregate* is an object that is made up of other objects

Therefore aggregation is a *has-a* relationship

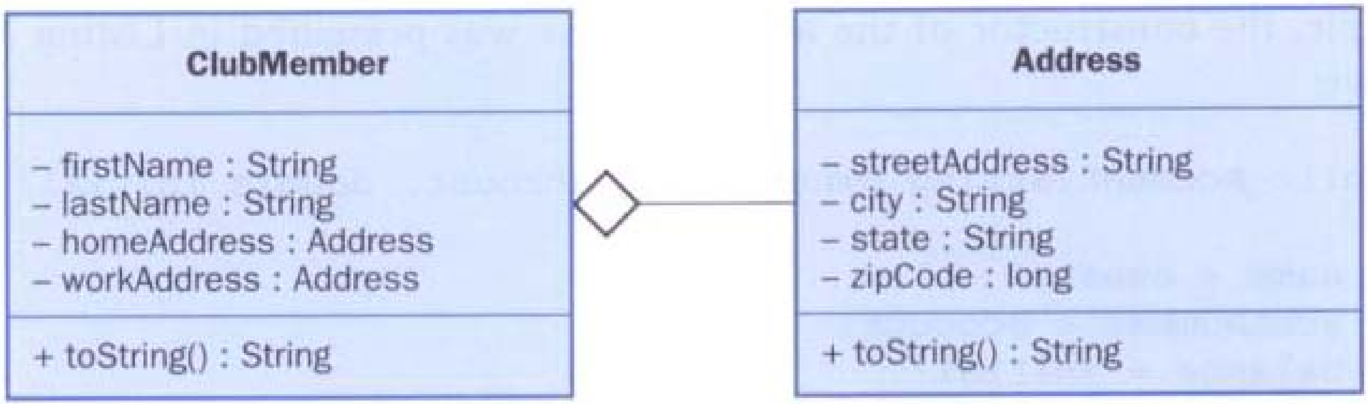
- A car *has a* chassis

In software, an aggregate object contains references to other objects as instance data

The aggregate object is defined in part by the objects that make it up

This is a special kind of dependency – the aggregate usually relies on the objects that compose it

Aggregation in UML



Reviews

Review – meeting of several people designed to examine a design document or section of code

Presenting a design or code causes us to think carefully about our work and allows others to provide suggestions

Goal of a review is to identify problems

Design review should determine if the system requirements are addressed

Defect Testing

Testing is also referred to as *defect testing*

Though we don't want to have errors, they most certainly exist

A *test case* is a set of inputs, user actions, or initial conditions, and the expected output

It is not normally feasible to create test cases for all possible inputs

It is also not normally necessary to test every single situation

Defect Testing

Two approaches to defect testing

- *black-box*: treats the thing being tested as a black box
  - Test cases are developed without regard to the internal workings
  - Input data often selected by defining *equivalence categories* – collection of inputs that are expected to produce similar outputs
  - Example: input to a method that computes the square root can be divided into two categories: negative and non-negative

Defect Testing

Two approaches to defect testing

- *white-box*: exercises the internal structure and implementation of a method.
  - Test cases are based on the logic of the code under test.
  - Goal is to ensure that every path through a program is executed at least once
  - *Statement coverage* testing – test that maps the possible paths through the code and ensures that the test case causes every path to be executed

Other Testing Types

Unit Testing – creates a test case for each module of code that been authored. The goal is to ensure correctness of individual methods

Integration Testing – modules that were individually tested are now tested as a collection. This form of testing looks at the larger picture and determines if bugs are present when modules are brought together

System Testing – seeks to test the entire software system and how it adheres to the requirements (also known as *alpha* or *beta* tests)

Regression Testing – seeks to verify that recent changes have not broken existing functionality. Typically a small subset of test cases designed to cover key areas of functionality.

Test Driven Development

Developers should write test cases as they develop their source code

Some developers have adopted a style known as *test driven development*

- test cases are written first
- only enough source code is implemented such that the test case will pass

Test Driven Development

Test Driven Development Sequence

1. Create a test case that tests a specific method that has yet to be completed
2. Execute all of the tests cases present and verify that all test cases will pass except for the most recently implemented test case
3. Develop the method that the test case targets so that the test case will pass without errors
4. Re-execute all of the test cases and verify that every test case passes, including the most recently created test case
5. Clean up the code to eliminate redundant portions (refactoring)
6. Repeat the process starting with Step #1

Debugging

Debugging is the act of locating and correcting run-time and logic errors in programs

Errors can be located in programs in a number of ways

- you may notice a run-time error (program termination)
- you may notice a logic error during execution

Through rigorous testing, we hope to discover all possible errors. However, typically a few errors slip through into the final program

A *debugger* is a software application that aids us in our debugging efforts

Simple Debugging using `cout`

Simple debugging during execution can involve the use of strategic **`cout`** statements indicating

- the value of variables and the state of objects at various locations in the code
- the path of execution, usually performed through a series of “it got here” statements

Consider the case of calling a method

- it may be useful to print the value of each parameter after the method starts
- this is particularly helpful with recursive methods

Debugging Concepts

Formal debuggers generally allow us to

- set one or more *breakpoints* in the program. This allows to pause the program at a given point
- print the value of a variable or object
- step into or over a method
- execute the next single statement
- resume execution of the program

Key Things to take away:

- You tell me! ☺

References:

1. Walter Savitch, Problem Solving with C++. Pearson, 9th Edition, 2014, ISBN 978-0-13-359174-3
2. T. DeMarco, *Structured Analysis and System Specification*, 1979, ISBN 978-0-13-8543808
3. T DeMarco, Structured Analysis, Structural Design and Materials Conference 2001, Software Pioneers, Eds.: M. Broy, E. Denert, Springer 2002 http://cs.txstate.edu/~rp31/papersSP/TDMSpringer2002.pdf
4. Stevens, W., G. Meyers, and L. Constantine, *Structured Design*, IBM Systems Journal, Vol 13, No 2. 1974
5. Fairley, Richard E., *Software Engineering Concepts*, McGraw-Hill, 1985, ISBN 0-07-019902-7