Object-Oriented Programming

CSCI-UA 0470-001
Class 4
Instructor: Randy Shepherd

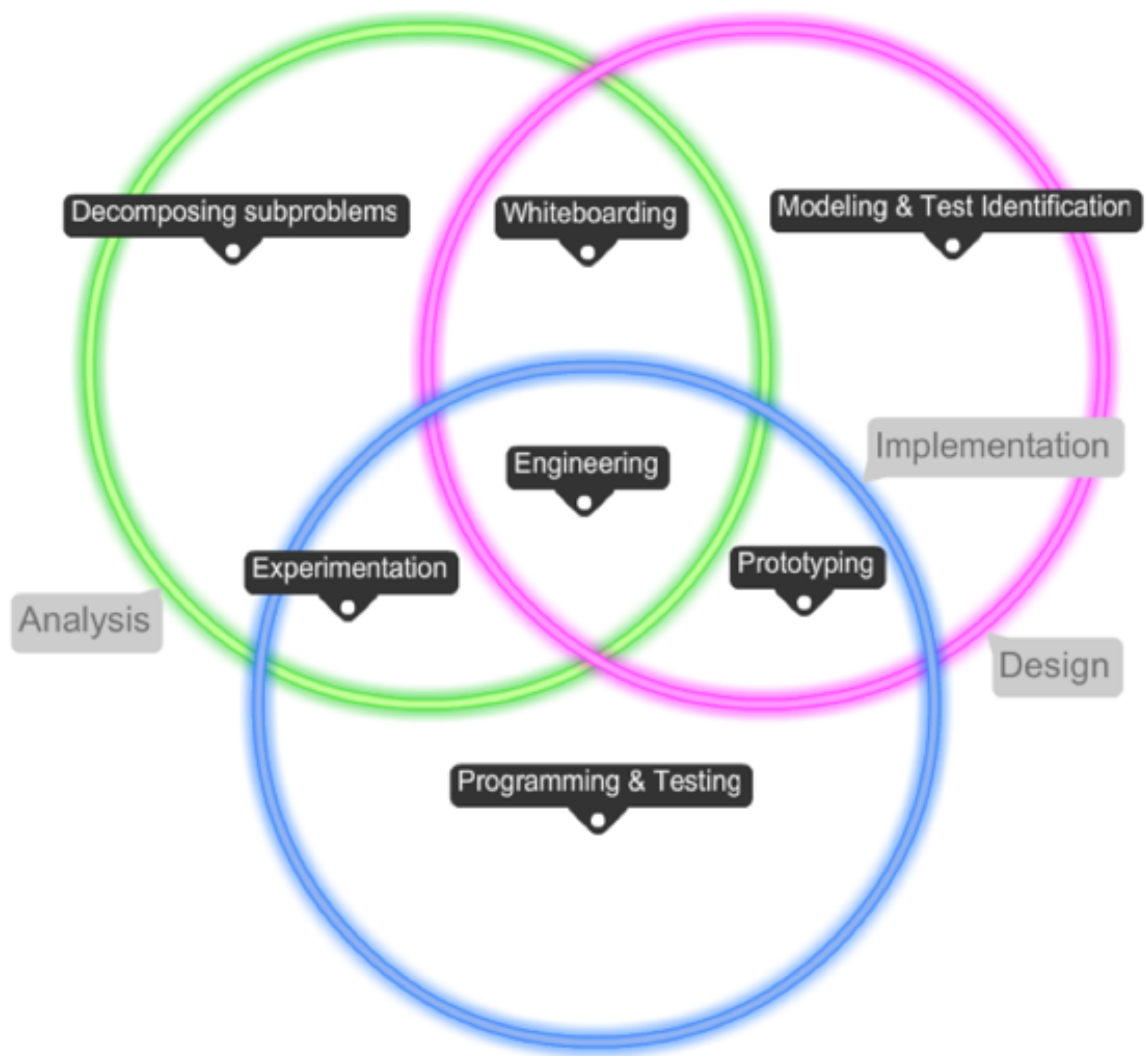
Object-Oriented Analysis and Design

Analysis, Design, Implementation

- Non-trivial software development requires forethought. Jumping into an editor first thing will get you into trouble.

- The book calls Analysis, Design & Implementation 'phases' and treats them as sequential. This is naive and not the way things work in practice.

Iterative A/D/I



Analysis

- Think the problem through, get an understanding in broad strokes of what you will be building.
- Decompose the problem into subproblems.
- Identify the ‘nouns’ and the ‘verbs’.

Design

- Introduce the design phase…
- Are there design patterns you can use to solve the problem?
- Model some classes based on your ‘nouns’ and ‘verbs’
- Continue to analyze. Looking for opportunities to add constraints that eliminate complexity.

Implementation

- Decide how to distribute work. Who works on what subproblem?
- Start coding!
- …but continue to refine you understanding of the problem and your design approach.

Design Patterns

(adapted from UofW CSE403: Software Engineering)

What are Design Patterns?

- is a general, reusable solution to a commonly occurring problem
- is abstract from programming language
- identifies classes and their roles in the solution to a problem
- patterns are not code or designs; must be instantiated/applied

Why are Design Patterns?

- patterns are a common design vocabulary
- allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation
- patterns capture design expertise and allow that expertise to be communicated
- promotes design reuse and avoid mistakes
- improve documentation (less is needed) and understandability (patterns are described well once)

Ex. Iterator Pattern

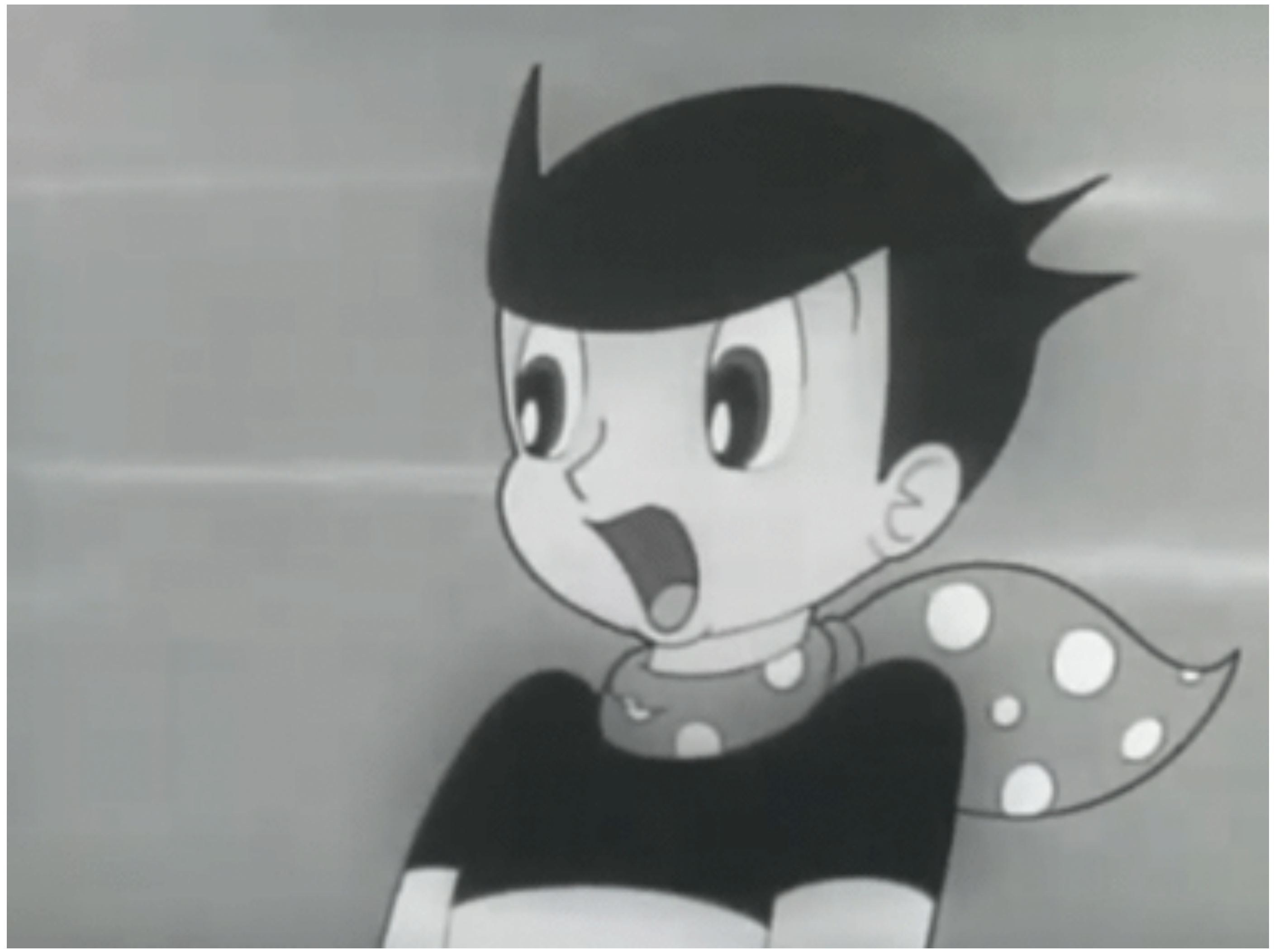
- *iterator*: an object that provides a standard way to examine all elements of any collection
- uniform interface for traversing many different data structures
- supports concurrent iteration and element removal

```java
for (Iterator<Account> itr = list.iterator(); itr.hasNext(); ) {
    Account a = itr.next();
    System.out.println(a);
}
```

```
set.iterator()
map.keySet().iterator()
map.values().iterator()
```

Ex. Chain of Responsibility

- a design pattern consisting of a source of ‘command’ objects and a series of ‘processing’ objects
- Each processing object contains logic that defines the types of command objects that it can handle;
- A processing ‘pipeline’, good for programs that pass the same data through a series of ‘phases’.



Confused? Lets look at some code.

https://github.com/nyu-oop/chain-of-responsiblity

Project Intro

Project Intro

- Translator from a subset of Java to a subset of C++
- Primary tool in this effort is a tool called Xtc, it has many features, you need to learn how to use them.
- Also, a supporting toolchain with Sbt, Junit, Git, etc.
- We will look closer at Xtc and the toolchain in next week

Source Language

- The source language is the language of programs serving as inputs to the translator; it is a restricted version of Java.
- Java *without* nested classes, anonymous classes, interfaces, enums, annotations, generics, the enhanced forloop, varargs, boxing/unboxing, abstract classes, synchronized methods and statements, strictfp, transient, volatile, lambdas, etc.
- Whats left?
    - Primarily interested in modeling basic translation and dynamic dispatch at first
    - Therefore omit method overloading but not method overriding for the first version
    - Other basic language features, static methods, packages, etc.
- You will be provided with test inputs to test your translator that exercise the features of source language.
- You can assume that source programs are free of compile-time errors.

Source Language

- Java includes an extensive set of platform libraries, with the facilities in the java.lang package

- For our translator, we only support..

  - the hashCode(), equals(), and toString() methods in java.lang.Object

  - printing of numbers and strings through out.print() and out.println() in java.lang.System.

- Furthermore, we do not support the dynamic loading of Java classes, but rather translate all Java classes of the source program into one C++ program.

Target Language

- The target language is the language of outputs of the translator; it is a restricted version of C++
- C++ without virtual methods, inheritance, templates, lambda abstractions, auto, decltype, etc
- Whats left?
  - basic classes, exceptions and namespaces
  - “C with classes”
  - For the first version of the translator, you can ignore memory management

Translator Language

- Java 1.7
- You should make extensive use of the OO structures available in Java. (Abstract Classes, Interfaces, etc.)
- You should leverage good OOP design principles. (Encapsulation, Polymorphism, Inheritance)
- You should leverage design patterns discussed in class. (Chain of Responsibility, Decorator, etc.)
- You should make use of all the libraries available to you. (Xtc, Junit, Logback, etc.)
- You should use good software engineering practices. (TDD, Pair Programming, Code Reviews, etc.)

How do we start?

- CRC cards exercise, at the end of class today.
- We also need some testing strategies:
  - Java programs that are test inputs and use translator.
  - Compile and run C++ output; then compare.
  - Write unit tests for small pieces of the translator
- We need to decompose the project into subproblems!

Phased Approach

- The translator will have several ‘phases’.
- Incremental steps towards goal.
- Allows for “divide and conquer” approach in your team.
- Phases can be worked on independently, to an extent.
- These are our “subproblems”.
- Additional phases will be added in the second half of semester.
- Before we can talk about what the phases are we need to understand an important concept….

Source Representation

- Compilers first create a ‘parse tree’ aka ‘concrete syntax tree’.
- A tree that represents the actual source code (expressions, statements, etc.).
- Contains the ‘tokens’ of the language, tied closely to the ‘grammar’ of the language.
- For us, what is important for translation are the *semantic constructs* of the language. Not the syntax itself.
- Concrete Syntax / Parse trees not right choice for our translator….

Abstract Syntax Tree

- An abstract syntax tree (AST) is a tree representation of abstract syntactic structure of a program.
- Each *node* of the tree denotes a construct occurring in the source code.
- The syntax is "abstract" in not representing every detail appearing in the real syntax.

Example

```
while b ≠ 0
    if a > b
        a := a – b
    else
        b := b – a
return a
```

Why Abstract Syntax Tree?

- An AST can be edited and enhanced with information on each node. Such editing is impossible with the source code of a program, since it would imply changing it.
- An AST usually contains extra information about the program, due to the *consecutive phases of analysis.*
- Traversing an AST is done many times during translation. Making that process convenient is important, so it often makes sense to use the Visitor Pattern.
- We’ll cover the Visitor Pattern in this class.

Abstract Syntax Tree Types

- Statically-typed AST
  - You define types that map to nodes of the AST.
  - Figuring out how to map a source node type to a target node type is the translation approach.
  - You can have all phases in one class.
  - Data kinds are fixed with this approach but there will be more phases in the second half. Trickier to evolve elegantly.
- Dynamically-typed AST
  - All nodes of the same type (xtc.lang.GNode)
  - Trade-off: Flexibility for safety. Normally I would vote for safety, but we need to move quickly.
  - Enables a class per phase, much easier to add a new phase in the second half.
- Xtc supports both, but we will write our translator with the dynamically-typed AST.

Project Phases

Phase 1
Load all sources as Java AST

- Input: Source Java File
- Steps:
  - Load source file, generate Java AST.
  - Load dependency classes that are in scope, eg. other classes in package and imports
  - Generate Java AST for dependencies
- Output: Set of ASTs representing all Java classes to be translated.
- (Xtc and JavaFiveImportParser do most of the work for this phase)

Phase 2
Generate AST for inheritance hierarchy

- Input: Set of output nodes from phase 1
- Steps:
  - Traverse set of nodes from step 1
  - For each node, build an ast that represents the data layout and vtable
  - layout that will be in the C++ header file.
- Output: A set of nodes representing the vtable and data layout
- (You will design the AST schema before writing the code for this.)

Phase 3
Write C++ header with inheritance hierarchy

- Input: Set of output nodes from phase 2
- Steps:
  - Traverse set of output nodes from step 2
  - For each node, print into the C++ header file that contains all the class definitions.
- Output: No output. However, the output.h file is on disk and complete at this point.

Phase 4
Mutate/Decorate Java Ast to C++ Ast

- Input: Set of output nodes from phase 1
- Steps:
    - Traverse set of nodes from step.
    - For each node, mutate it and translate to a node representing C++ code.
    - You will need to mutate and generate additional nodes.
- Output: Set of nodes representing the C++ implementation code to be printed

Phase 5
Write C++ implementation files

- Input: Set of output nodes from phase 4
- Steps:
  - Traverse set of nodes from phase 4
  - For each node, print into the C++ implementation file that contains all the class definitions.
- Output: No output. However, the output.cc file is on disk and complete at this point.

UML

What is UML?

- UML stands for Unified Modeling Language
- UML is a diagramming language designed for Object-Oriented programming
- UML can be used to describe:
  - the organization of a program
  - how a program executes
  - how a program is used
  - how a program is deployed over a network
  - …and more

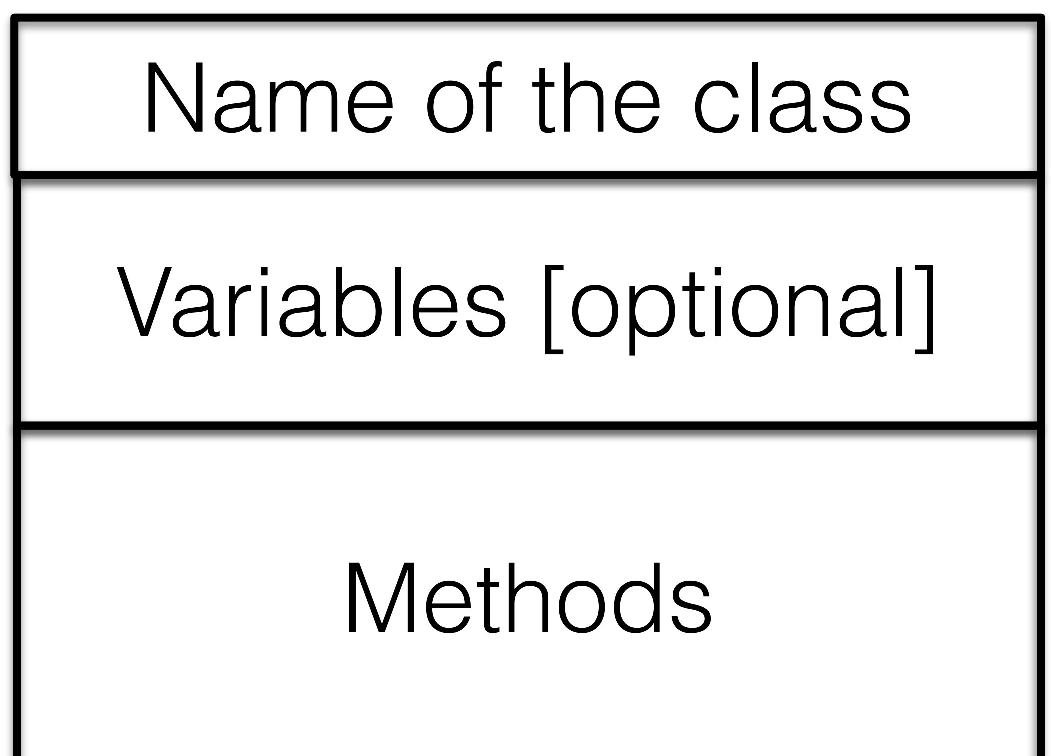
Why is UML?

- UML is a big, complicated diagramming language, most of which almost nobody uses. Really.
- UML comprises at least seven or eight different kinds of diagrams
- A class diagram is often all that is needed
- UML is a means to an end for us, we only want to have a vocabulary for talking about class relationships and definitions
- We'll cover most of classes and a few of the most important relationships

Classes

- A class diagram shows classes, interfaces, and their relationships

- A class is drawn as a rectangle with two or three compartments:



Variables

- A variable is written as:

  visibility name : type

  where:

  - \+ means public visibility
  - # means protected visibility
  - \- means private visibility
  - <blank> means default (package) visibility

- Example: +length:int

Variables

- Static variables are underlined
- An initial value can be shown with =value
- Example:
  -<u>numberOfEmployees:int</u>=10

  means numberOfEmployees is:
  - private
  - static
  - integer
  - and has 10 as its initial value

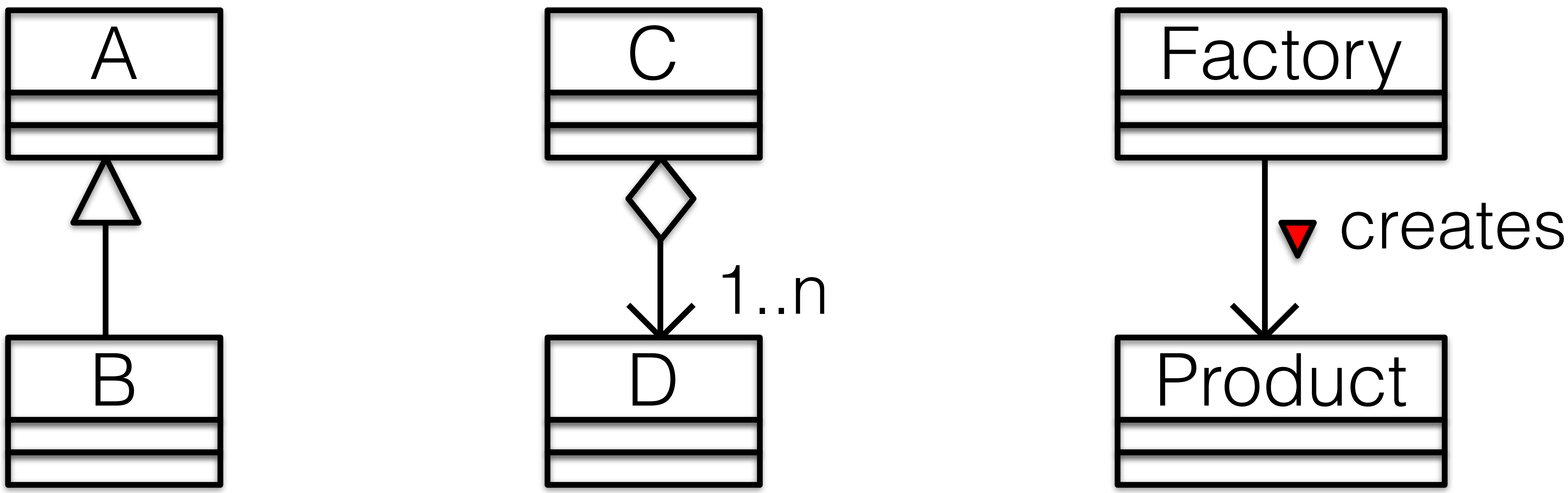
Methods

- Methods are written as:
  visibility name (parameters) : returnType
  where:
  - visibility uses the same syntax variables (+, -, #, blank)
  - parameters are given as name:type
  - if the returnType is void, it is omitted
  - constructors are preceded by «constructor»
  - interfaces are preceded by «interface»
  - an ellipsis (…) indicates omitted methods

Example: Class Diagram

Card
cardId:int -copy:boolean=false
«constructor» Card(int id) +isKind(desiredKind:int) +isSharable():boolean

Class Relationships



Class B extends class A

Class C contains 1 to n objects of class D

Other kinds of relations

CRC Cards

(Adapted from Zenebe & Miao, 2001)

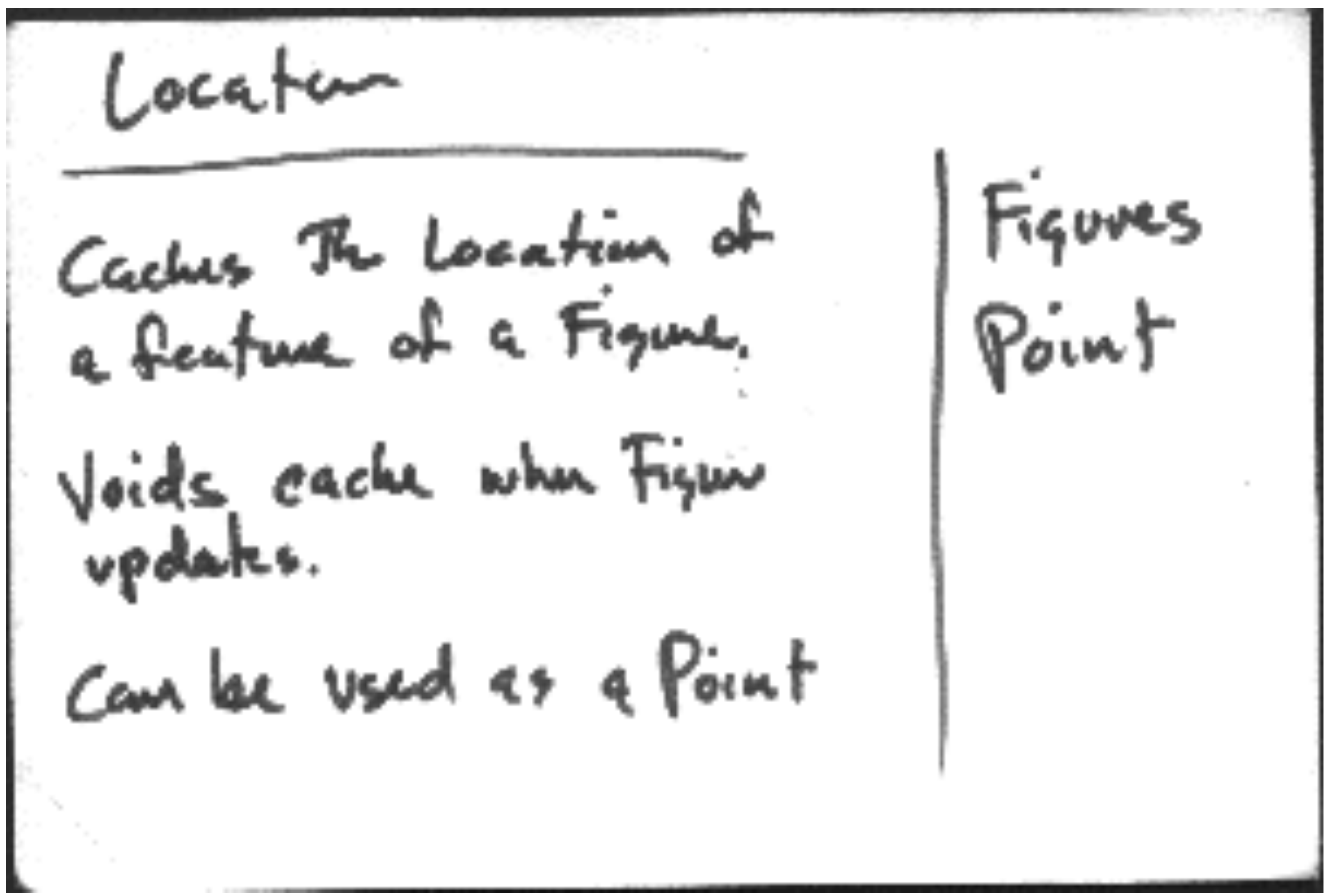
CRC Cards

- Class-responsibility-collaboration cards are a brainstorming tool used in the design of object-oriented software
- An informal approach to OO modeling
- The card is partitioned into three areas
  - On top of the card, the class name
  - On the left, the responsibilities of the class
  - On the right, collaborators (other classes) with which this class interacts to fulfill its responsibilities
- Useful when working through design in teams

CRC Team

- The ideal team size is about 4-6 people. What a coincidence!
- Traditionally there are a number of roles, we'll simplify. We only need a facilitator and a scribe.
- Group members create, supplement, stack, and wave cards during the walk-through of scenarios.

Example CRC Card



CRC Session

- Use physical index cards. Better for group discussion.
- Session includes physical simulation of the system using the cards a representations of the collaborating entities
- All ideas are potential good ideas, keep a log of ideas ‘to keep for later’

CRC Process

- Step 1) brainstorm
  - What are the ‘nouns’, what are the ‘verbs’
- Step 2) identify classes
  - Throw a bunch of ideas at the wall and then as the conversation processes, eliminate ones you do not need
- Step 3) Scenario execution
  - Role-play the execution of the main functions of your application. This is the most important part!

CRC Session Tips

- Start simple!
- Name things well.
- Write out descriptions so everyone understands the role of each class
- Lay the cards out to see how they might interact
- Don’t get too attached to any idea, be prepared to throw ideas away when a better one comes along!

Lets try…

- We’ll do a CRC session on the translator based on the phases we discussed in class.
- Don’t worry about getting it perfect, you don’t really know that much yet. This will be a good way to identify questions you need answers to.
- Look for opportunities to apply the principles of OOP; encapsulation, inheritance and polymorphism
- Chain of responsibility would be a great pattern here and should get you started with your cards
- Break into your project teams…

Teams!