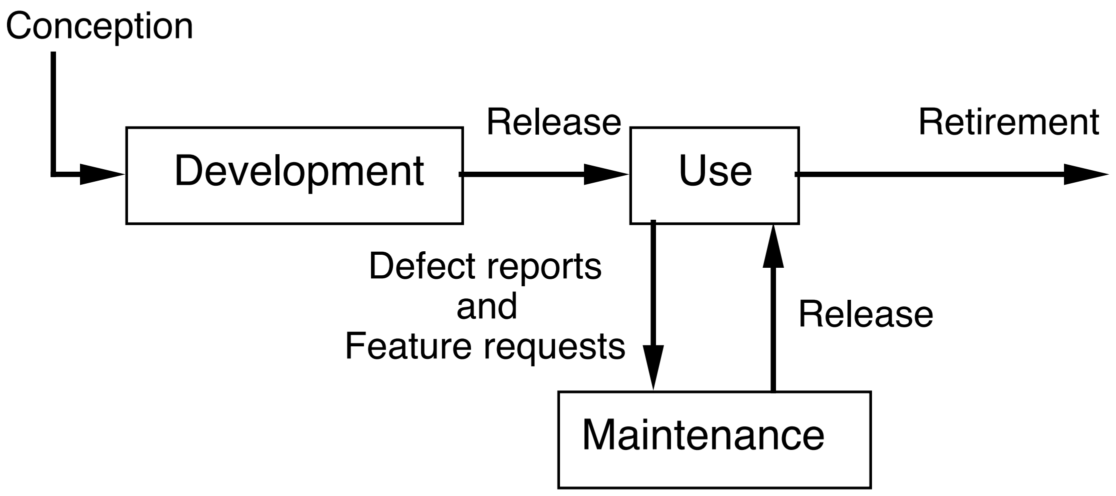
Software Development: An Introduction

Fact: *Software is hard*.

Imagine the difficulty of producing Windows 2000

- 29 million lines of code
- 480,000 pages of listing if printed
- a stack of paper 161 feet high
- Estimates were made that there were 63,000 bugs in the software when it was released.

Software Life Cycle: Stages that a program goes through.



Development: Months or a few years

Maintenance: Modify program to enhance it or to fix problems

Use and Maintenance: Possibly many years

Problem

Maintenance will probably be done by programmers who did not develop the original program.

Goal: Construct programs that are easy to read, understand, and modify.

Quote: Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. — John F. Woods

Qualities of Good Software

Correctness
- Performs the task defined by the specification.

Robustness
- Reacts appropriately to unusual conditions.
- Degrades gracefully under adverse conditions.

Extendibility
- Allows changes and extensions of the specification easily.

Reusability
- Be usable, at least in part, in the construction of other programs.

Compatibility
- Works well with other, preexisting software that it uses.

Efficiency
- No excessive use of resources (time and memory).

Usability
- Useful to many without special knowledge.

Functionality
- Provides all the behavior required to solve the problem and many similar ones.

Timeliness
- Completed on time.

Style
- Has aesthetic properties that make it enjoyable to maintain.

Development Process

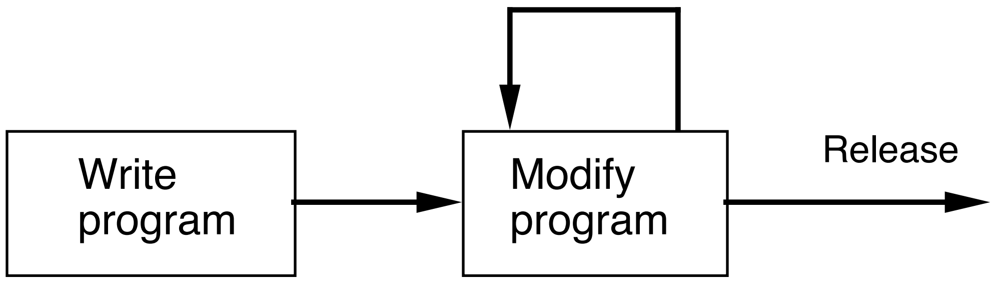
- Establish requirements clearly.
- Plan design of program carefully.
- Follow design, coding, and documentation guidelines.

Changes made late in the life cycle are much more expensive than those made early.

Putting an extra effort into program development will likely reduce the building and maintenance costs significantly.

Build-and-Fix

How we create small programs.

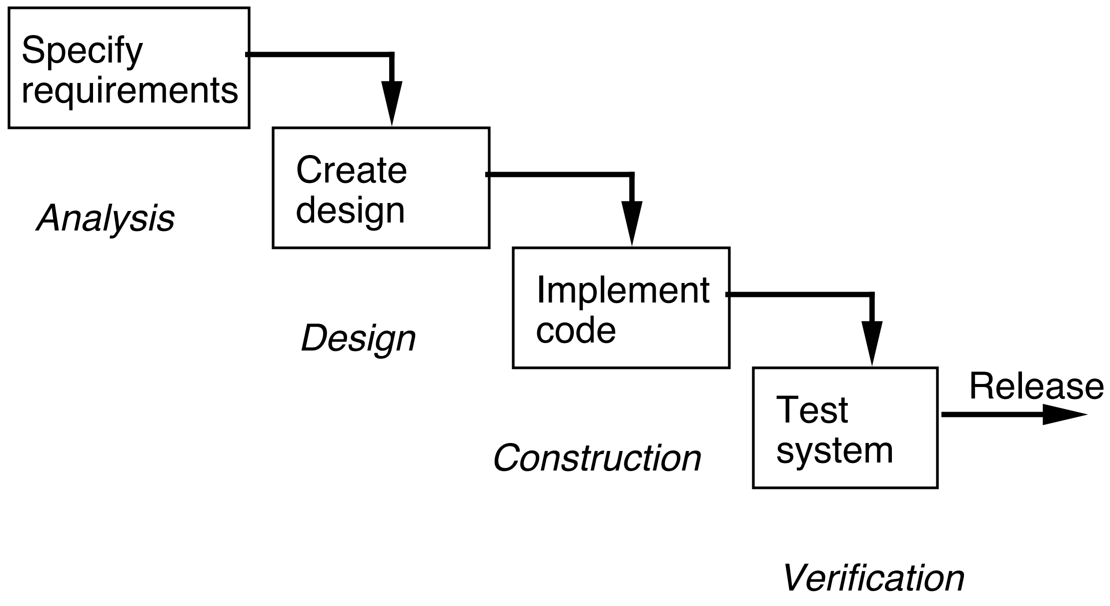


Programmer simply reacts to problems.

- Ad-hoc strategy
- Not really a development methodology.
- Program not designed to aid debugging and maintenance.

Methodology I: Waterfall Model

- Popular in the 1970's.
- Linear sequence of stages.



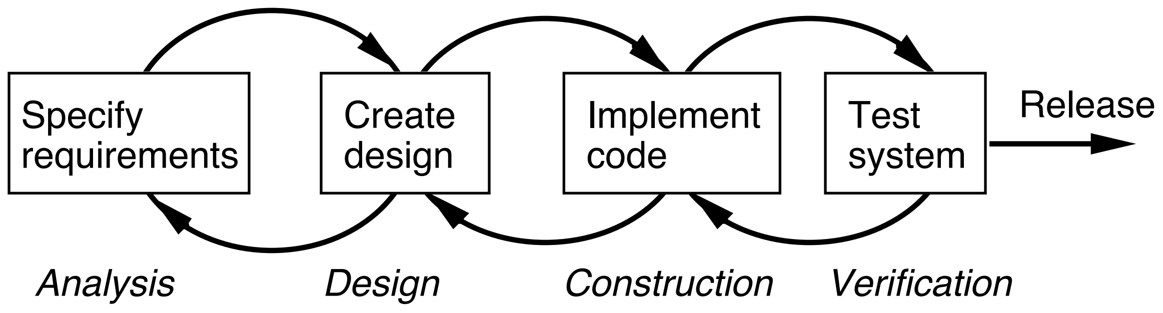
- No provision for revisiting earlier stages.
- Each stage must be completed accurately before going on to the next.

- **Problem**: What if an error is uncovered in a later stage that depends on a mistake made in an earlier phase?

Methodology II: Iterative Models

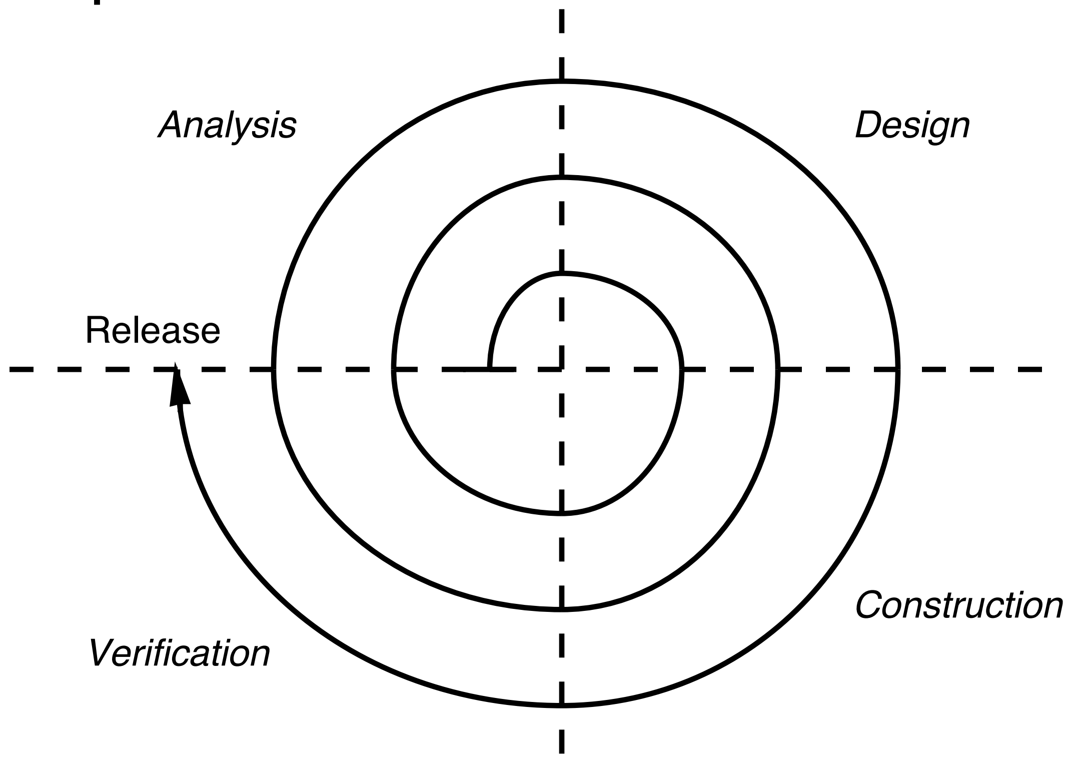
- Cycle through phases, permitting a revisit of earlier ones.

Modified Waterfall



- Each stage should be developed as completely as possible.
- Backtracking should be used to correct errors or problems uncovered in later phases.
- Repeat the entire process creating a more complete implementation each time.

Simple Spiral



Spiral process

- Starting at middle, go through successive requirement analysis, design creation, code implementation, and evaluation phases.
- Number of iterations is arbitrary: Do as many as you need.

Prototypes

- Programs that implement part of a system to explore possibilities.
- Ignore efficiency and completeness.
- Help designer avoid misunderstanding requirements.
- Use “stubs” for missing units (classes and methods), that is, empty code that compiles.
- Prototypes may be disposable or evolutionary (develop into final program).

Analysis

- Investigate the problem.
- Do not define a solution yet.
- Specify requirements.
- Use “use case” methodology.
- Analyze the domain of the problem.
- Create conceptual model, identifying concepts and relationships.
- Do not define software components.
- Represent concepts in the real-world problem domain.

Design

- Construct a logical solution that satisfies the requirements.
- Assign responsibilities to the various components of the solution (the most important task).
- Identify suitable classes and objects.
- Use class diagrams (relations among classes) and collaboration diagrams (objects with flow of messages between them).

Construction

- Implement the design in a programming language.

Requirement Specifications

Requirements must be specified correctly before the problem can be analyzed.

Main Problem: Understanding the needs of the users.

Example: Gradebook

Overview Statement

The purpose of this project is to provide a record keeping system that maintains and analyzes the grades for a course.

Customers

Instructors of courses.

Goals

The goal is to provide an accurate and convenient way to maintain grades for a course.

Specific goals

- Flexible tools to create and alter the roster.
- Classify course grades to allow weighting of course work.
- Persistent storage to hold the grades during a semester.

System Functions

Basic Functions

R1.1	Create a roster of student names and ID numbers.
R1.2	Add a student to the roster.
R1.3	Delete a student from the roster.
R1.4	Print a listing of students with the grades.
R1.5	Provide persistent storage mechanism to save and retrieve rosters.

Grade Functions

R2.1 Classify and accept one set of grades for the roster of students.

R2.2 Allow three classifications for grades: Exams, Projects, and Homework/quizzes.

R2.3 Change a grade.

Analysis Functions

R3.1 Compute total scores for students based on weight factors given to each category of grades.

R3.2 Sort list of students by their total grades.

R3.3 Print students in the order of their total scores.

R3.4 Print histogram of total scores.

UML: Unified Modeling Language

A standard modeling *notation*.

Not a development process.

Authors Grady Booch
Jim Rumbaugh
Ivar Jacobson

UML Diagrams

Use case diagrams

Static structure diagrams
- Conceptual model
- Class diagrams
- Object descriptions

Interaction diagrams
- Sequence diagrams
- Collaboration diagrams

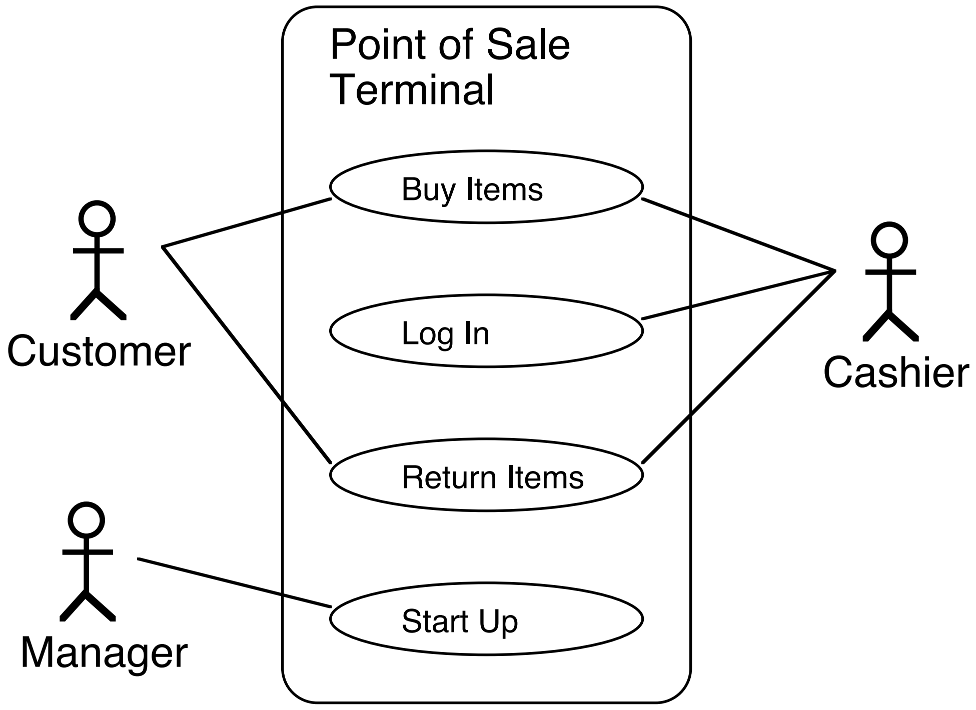
State diagrams
- State diagrams
- Activity diagrams

Implementation diagrams
- Package diagrams
- Component diagrams
- Deployment diagrams

Use Cases

Draw a diagram showing the interactions between “actors” in the system as classified by use cases.

- Show boundary between system and “users”.
- Use cases are shown as ovals.
- Actors are shown as stick figures joined to the use cases in which they participate.



An actor is a user of the system in a particular role.

A user can be a person, another system, or a hardware device.

Created by Ivar Jacobson.

Narrative Description

Describe the steps in the use case in a structured prose format.

- Identify the initiator of the use case.
- Describe what the actor(s) do and how the system responds as the use case plays out.
- Specify alternative courses that result from unexpected occurrences.

Use Cases in Software Development

Iterative development cycles are organized by use case requirements.

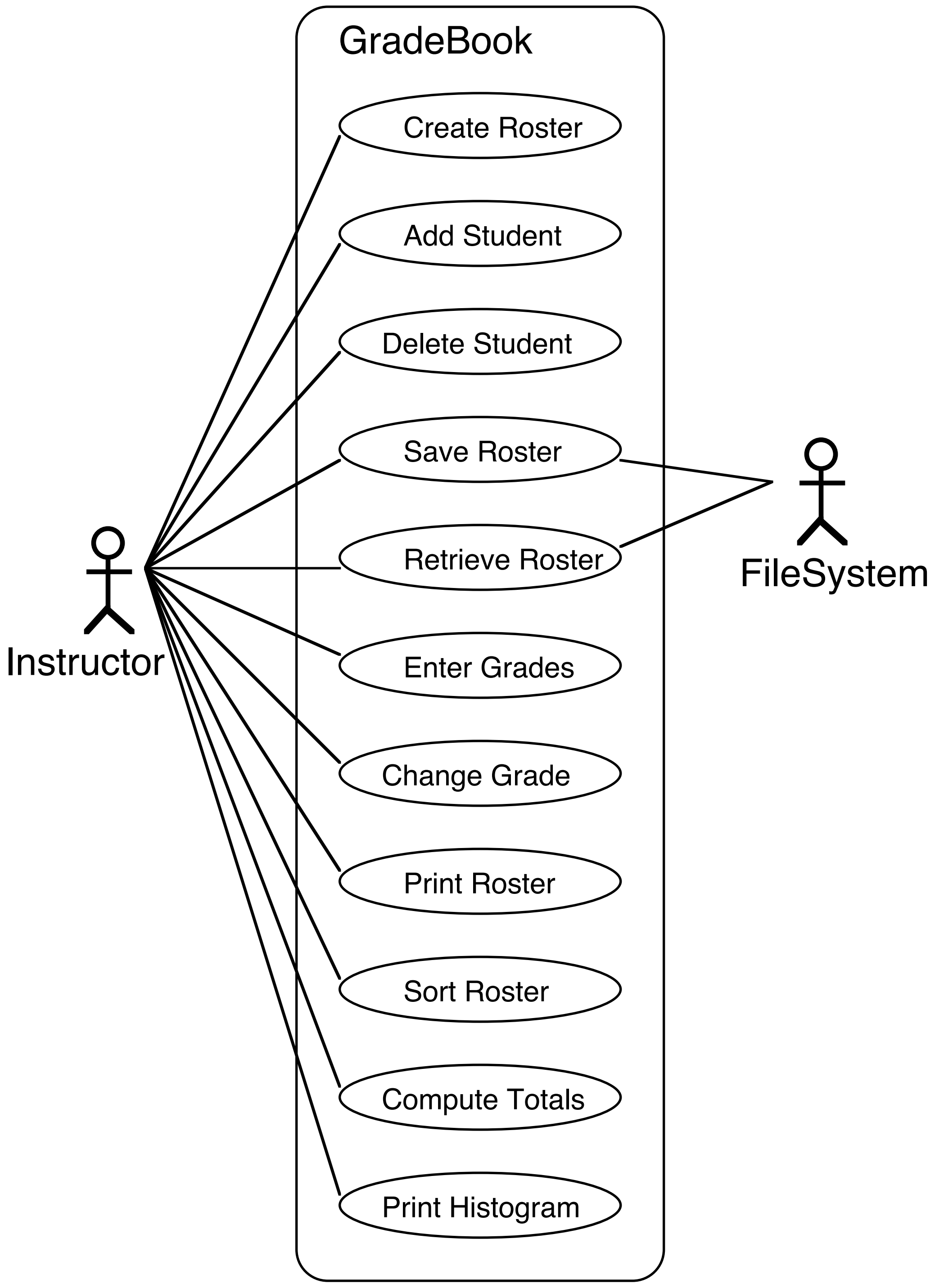
Use cases should be ranked with high-ranking cases tackled in early development cycles.

Use cases can be developed at different levels of detail working from high-level use cases to more detailed (expanded) ones as the development proceeds.

Caveat

Some OO designers feel that a use-case analysis leads to non-object-oriented analysis.

Example: GradeBook



Use case: Create Roster

Actors: Instructor (initiator)

Purpose: Instructor enters the students of a class to form a new roster.

Overview: Instructor asks to create a new roster. Instructor enters name and ID for each student in a class to make the roster.

Typical Course of Events

Actor Action	System Response
1. This use case begins when Instructor signals the need for a new roster.	
	2. Requests name of class.
3. Instructor provides a name for the class.	
	4. Prompts for name and ID for each student.
5. Instructor enters each name and ID in the new roster.	
6. On completion of entries, Instructor indicates the end of the list.	
	7. Acknowledges existence of a new roster.

Alternative Courses

- Line 1: The current roster has not been saved. Indicate error.
- Line 6: No names entered. Indicate error.

Use case: Enter Grades

Actors: Instructor (initiator)

Purpose: Instructor enters the grades for an exam, project, or homework/quiz.

Overview: Instructor asks to enter a set of grades. Instructor enters a grade for each student in the roster.

Typical Course of Events

Actor Action	System Response
1. This use case begins when Instructor signals the need to enter a set of grades.	
	2. Requests the nature of the set of grades.
3. Instructor provides the nature: exam, project, or homework/quiz.	
	4. Requests the maximum score.
5. Instructor enters the maximum score.	
	6. Prompts with name of each student in roster.
7. Instructor enters the grade for each student.	
	8. Acknowledges recording of the set of grades.

Alternative Courses

- Line 1: No current roster. Indicate error.
- Line 7: A grade entered exceeds maximum. Request another.

Use case: Save Roster

Actors: Instructor (initiator), FileSystem

Purpose: Save the roster and grades to the file system.

Overview: Instructor asks to save the current roster.

Typical Course of Events

Actor Action	System Response
1. This use case begins when Instructor signals the need to save the current roster.	
	2. Saves current roster to the file system.
	3. Acknowledges that the current roster has been saved successfully.

Alternative Courses

- Line 1: No current roster. Indicate error.

Use case: Sort Roster

Actors: Instructor (initiator)

Purpose: Rearrange roster to be in sorted order.

Overview: Instructor asks that roster be sorted by name, by ID, or by total score.

Typical Course of Events

Actor Action	System Response
1. This use case begins when Instructor signals the need to sort roster.	
	2. Requests criterion for sorting.
3. Instructor answers by name, by ID, or by total score.	
	4. Acknowledges that the current roster has been sorted.

Alternative Courses

- Line 1: No current roster. Indicate error.
- Line 3: Criterion not specified. Indicate error.

Static structure diagrams

- Conceptual models
- Class diagrams
- Object descriptions.

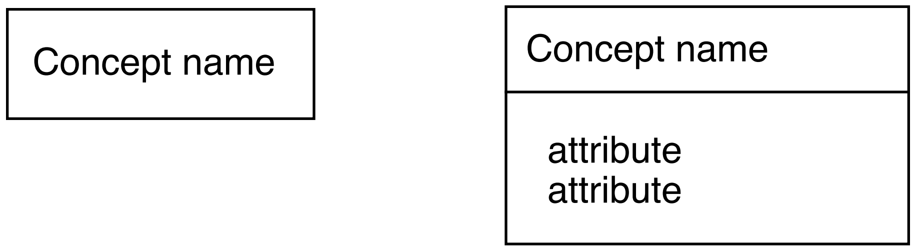
Conceptual Models

- Central to object-oriented design.
- Focus on domain concepts, not software.
- Decompose problem into individual concepts.

May show: Concepts
Associations between concepts
Attributes of concepts

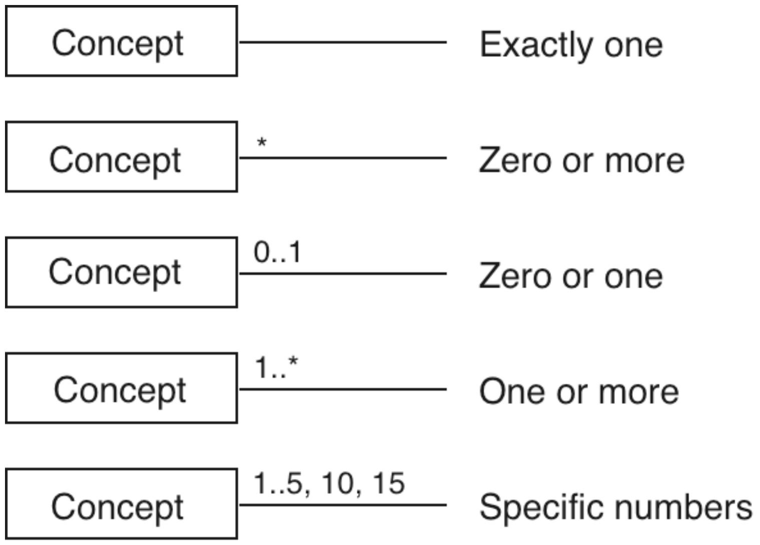
Concepts in Conceptual Models

Rectangles with one or two sections.



Concepts: physical objects, places, transactions, line items, containers of other things, abstract concepts, organizations, events, processes, records, financial instruments, catalogs, manuals, and so on.

Associations Solid line showing semantic relationships (multiplicity).



Developing a Conceptual Model

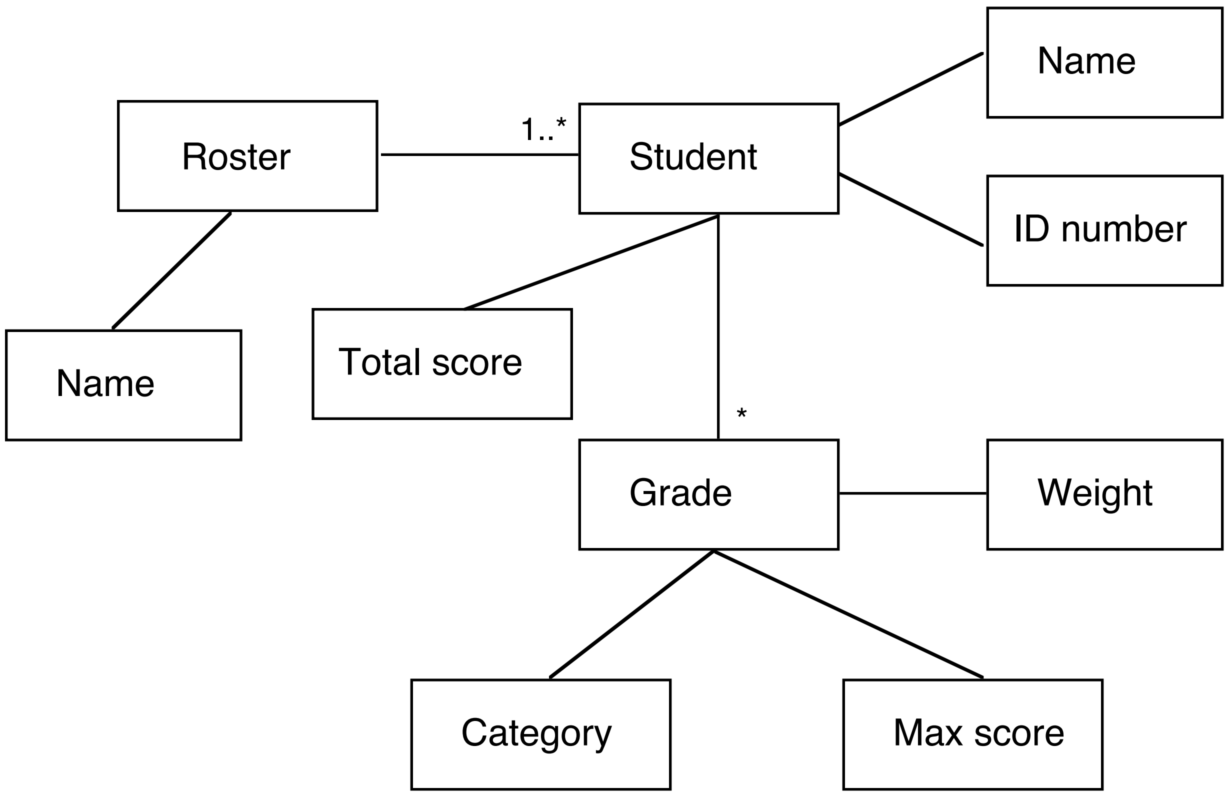
Identify the nouns and noun phrases in the requirements and the use cases that belong to the system.

GradeBook Example

roster, student, grade, name, ID number, position, listing, exam, project, homework, score, maximum score, weight factor, histogram, total score, set of grades, category of grades, sorting criterion.

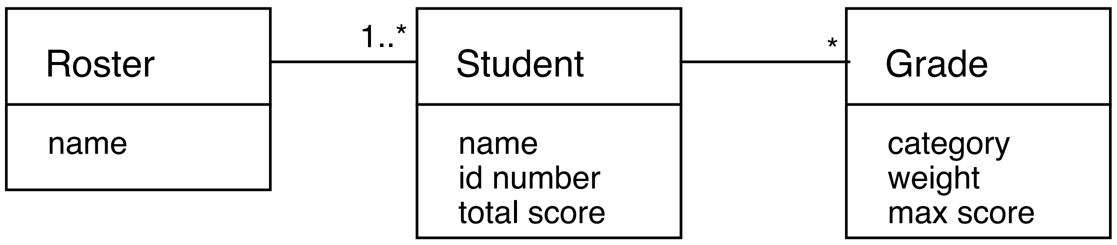
Discard nouns that are redundant, vague, an event or operation, outside the system, or an attribute.

Conceptual Model: Version 1



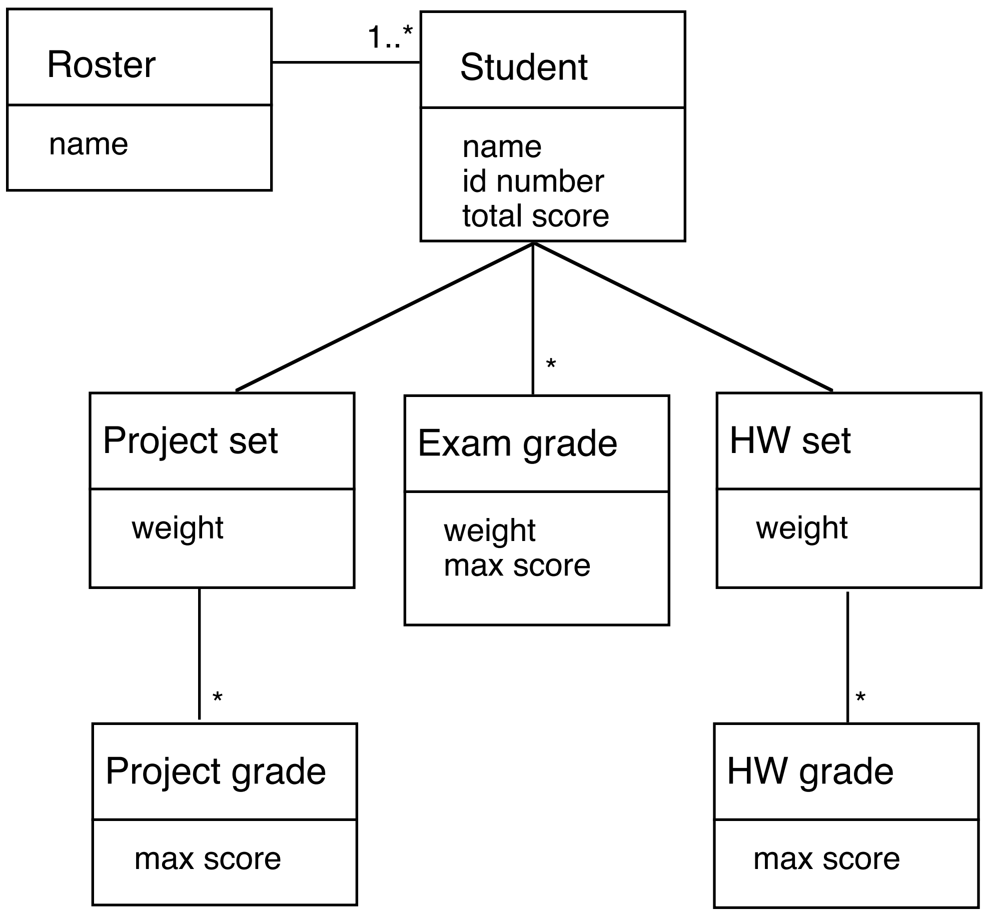
Some of these concepts should be attributes.

Conceptual Model: Version 2



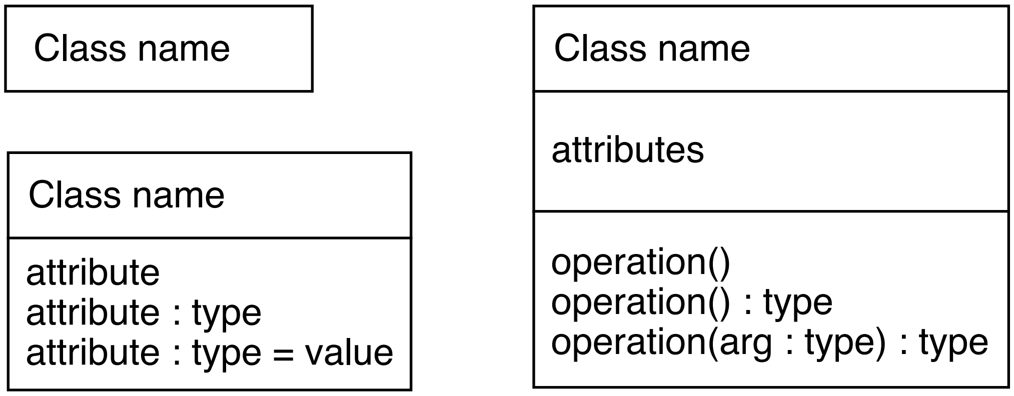
Categories of grades are handled differently. Split the concept.

Conceptual Model: Version 3

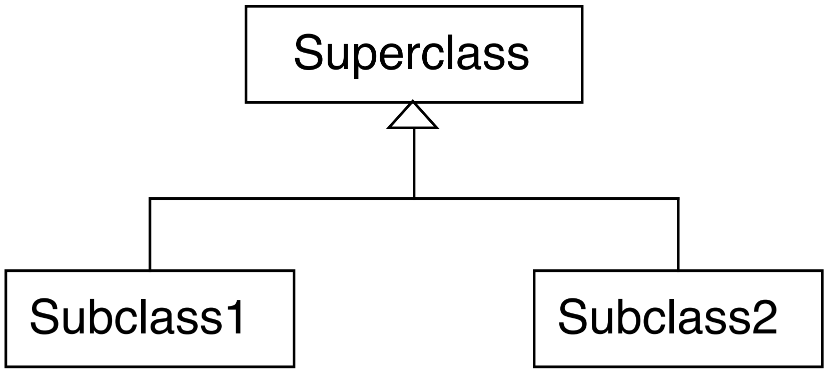


Class Diagrams

Rectangles of three forms.

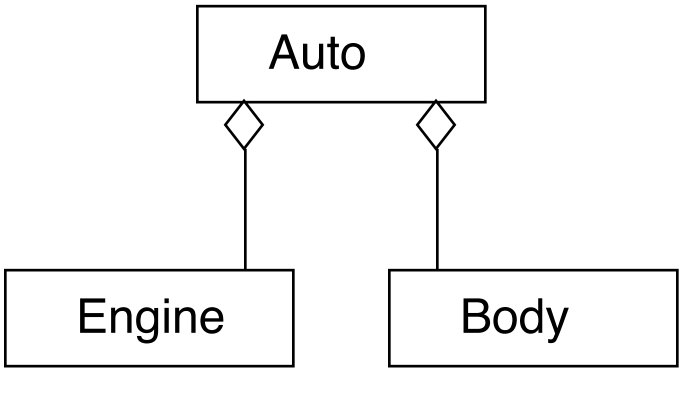


Inheritance



Composition

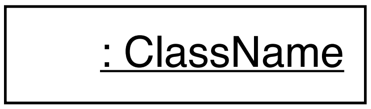
One class uses or "has-an" instance of another class.



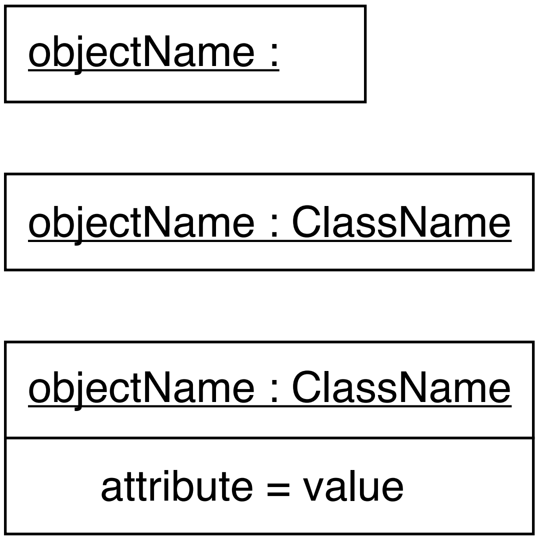
Object Diagrams

Rectangles with one or two sections and with the title underlined.

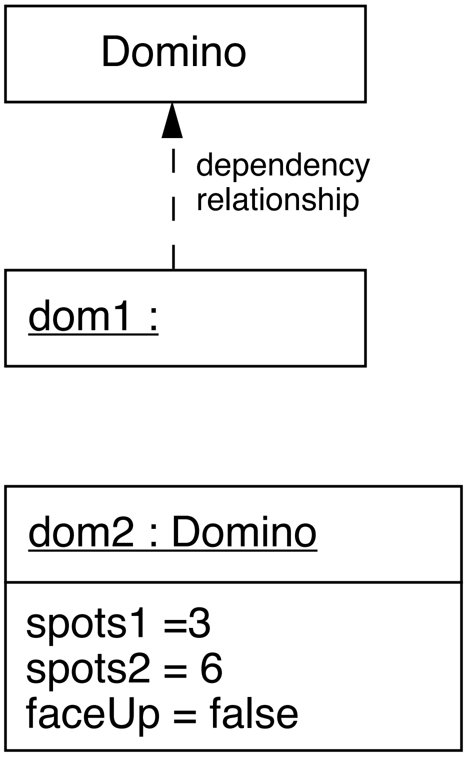
Generic object



Specific object

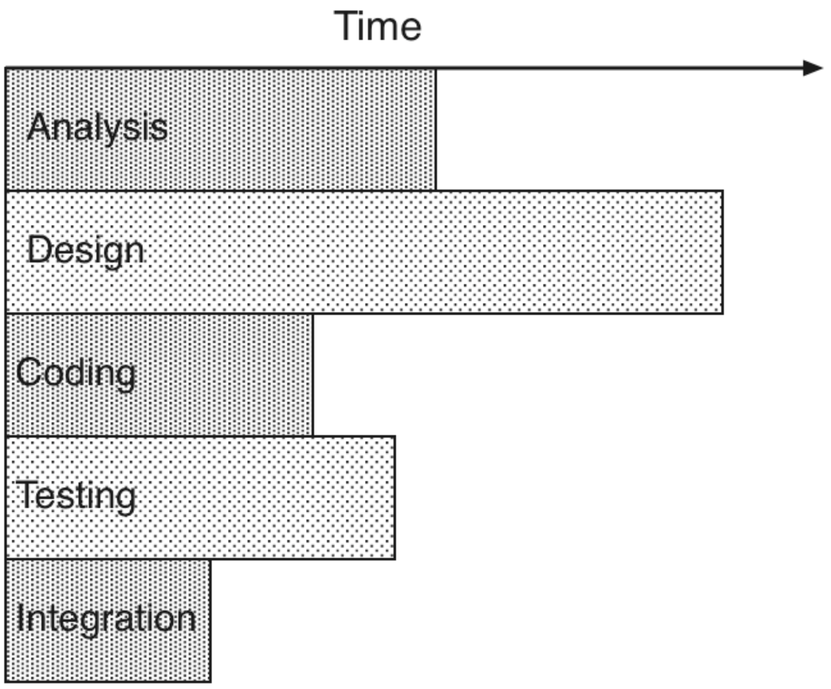


Instantiation of an object from a class:



Grady Booch Estimate

Time required for various activities in OO Development



Pair Programming

Pair programming is a style of programming in which two programmers work side by side at one computer, continually collaborating on the same design, algorithm, code, or test.

One of the pair, called the *driver*, is typing at the computer or writing down a design. The other partner, called the *navigator*, has many jobs, one of which is to observe the work of the driver, looking for tactical and strategic defects.

- Tactical defects are syntax errors, typos, calling the wrong method, and so on.
- Strategic defects occur when the driver is headed down the wrong path—what is implemented just won't accomplish what needs to be done.

The navigator is the strategic, long-range thinker. Any of us can be guilty of straying off the path, but a simple, "Can you explain what you are doing?" from the navigator can bring us back to earth.

The navigator has a much more objective point of view and can better think strategically about the direction of the work.

The driver and the navigator can brainstorm on-demand at any time.

An effective pair programming relationship is very active. The driver and the navigator communicate, if only through guttural utterances (as in, "Huh?"), at least every 45 to 60 seconds.

It is also very important to switch roles periodically between the driver and the navigator.

Benefits

1. **Quality**

   Pairs produce code with fewer defects

2. **Time**

   Pairs produce higher-quality code in about half the time as individual programmers.

3. **Morale**

   Pair programmers are happier programmers.

4. **Trust and Teamwork**

   Pair programmers get to know their teammates much better, which builds trust and teamwork.

5. **Knowledge transfer**

   Pair programmers know more about the overall system.

6. **Enhanced learning**

   Pairs continuously learn by watching how their partners approach a task, how they use language features, and how they develop programs.

Seven Habits of Effective Pair Programmers

Habit 1: **Take Breaks**

Since pair programmers keep each other focused and on-task, it can be intense and mentally exhausting.

At a minimum, each hour, stretch, and look at something more than three feet away.

Habit 2: **Practice Humility**

Excessive ego can prevent a programmer from considering other idea and can cause a programmer to become defensive when criticized.

Remember that the work is a team effort. Be ready to ask questions and to learn as well as teach.

Habit 3: **Be Confident/Be Receptive**

Try not to let insecurity or anxiety interfere with your work.

A fear of appearing stupid decreases the number of bold proposals and ideas that will be suggested.

Do not allow competition within the pair. Blame for problems or defects should never be placed on either partner.

Habit 4: **Communicate**

Communication between the pair is essential.

Think aloud so your partner can help you develop ideas.

Vocalize what you are doing as you do it.

Habit 5: **Listen**

Really listen to what your partner has to say before responding.

Don't assume you know what he or she is talking about or that he or she understands what you are trying to do.

Don't assume he or she is trying to insult you or to criticize your ideas.

Habit 6: **Be a Team Player**

Remember that your partner's work is your work. You are completely responsible for every thing your pair does.

Be alert and inquisitive. If you don't understand what's going on, ask.

If you can think of a better way, suggest it.

Habit 7: **Find a Balance between Compromise and Standing Firm**

The primary purpose of pairing is to work toward the best design possible, regardless of from whom the design originated.

Be willing to consider your partner's ideas, but don't always agree with everything your partner suggests.

For favorable idea exchange, there should be some healthy disagreement and debate.