Use a formal process & tools to facilitate and automate design steps:

- Requirements
- Specification
- System architecture
- Coding/chip design
- Testing

Text: Chapter 1.4
Other resources on course web page.
Object-Oriented Design

- Describe system/design as **interacting objects**
  - Across multiple levels of abstraction
  - Visualize elements of a design

- **Object** = state + methods.
  - **State** defined by set of “attributes”
    - each object has its own identity.
    - user cannot access state directly
  - **Methods** (functions/operations) provide an abstract interface to the object attributes.

- Objects map to system HW/SW elements
Objects and classes

• **Class**: an object [type](#) that defines
  • **state elements** for all objects of this type.
    • Each object has its own state.
    • Elements not directly accessible from outside
    • State values may change over time.
  • **methods** (operations) used to interact with all objects of this type.
    • State elements accessed through methods
Object-oriented design principles

• Some objects closely correspond to real-world objects.
  • Other objects may be useful only for description or implementation.

• **Abstraction**: list only info needed for a given purpose

• **Encapsulation**: mask internal op’s/info
  • Objects provide interfaces to read/write the object state.
  • Hide object’s implementation from the rest of the system.
  • Use of object should not depend on how it’s implemented
Unified Modeling Language (UML)

- Developed by Grady Booch et al.
  - Version 1.0 in 1997 (current version 2.4.1)
  - Resources (tutorials, tools): www.uml.org

- Goals:
  - object-oriented;
  - visual;
  - useful at many levels of abstraction;
  - usable for all aspects of design.

- Encourage design by successive refinement
  - Don’t rethink at each level
  - CASE tools assist refinement/design
UML Elements

- **Model elements**
  - classes, objects, interfaces, components, use cases, etc.

- **Relationships**
  - associations, generalization, dependencies, etc.

- **Diagrams**
  - class diagrams, use case diagrams, interaction diagrams, etc.
  - constructed of model elements and relationships

Free/open source UML diagramming tools are available
Structural vs. Behavioral Models

- **Structural**: describe system components and relationships
  - *static* models
  - objects of various classes

- **Behavioral**: describe the behavior of the system, as it relates to the structure
  - *dynamic* models
UML Diagram Types

- **Use-case**: help visualize functional requirements (user-system interaction)
- **Class**: types of objects & their relationships
- **Object**: specific **instances** of classes
- **Interaction diagrams (dynamic)**
  - **Sequence**: how sequences of events occur (message-driven)
  - **Collaboration**: focus on object roles
- **Statechart**: describe behavior of system/objects
- **Component**: physical view of system (code, HW)
- **Others** ....
UML use case diagrams

- Describe behavior user sees/expects ("what" – not "how")
- Describe user interactions with system objects
- Users = actors (anyone/anything using the system)

Example: Data acquisition system

- Translate to algorithms for system design
DAQ system use case description

- **User**
  - Select measure volts mode
  - Select measurement range or autorange

- **System**
  - If range specified
    - Configure to specified gain
    - Make measurement
      - If in range – display results
      - If exceed range – display largest value and flash display
  - If auto range
    - Configure to midrange gain
    - Make measurement
      - If in range – display mode
      - If above/below range – adjust gain to next range and repeat
      - If exceed range – display largest value and flash display
UML class (type of object)

Class diagram: shows relationships between classes
UML object

Object diagram: static configuration of objects in a system

- d1: Display
  - pixels: array[] of pixels
  - elements
  - menu_items

- Comment: pixels is a 2-D array
The class interface

- **Encapsulation**: implementation of the object is hidden by the class
  - **Interface**: How the user sees and interacts with the object
- **Operations** (methods) provide the abstract interface between the class’ implementation and other classes.
  - An operation can examine/modify the object’s state.
  - Operations may have arguments, return values.

- Often list only a subset of attributes/methods *within a given design context*
  - Those pertinent to that context
Choose your interface properly

- If the interface is too small/specialized:
  - object is *hard to use* for even one application;
  - even harder to *reuse*.

- If the interface is too large:
  - class becomes too *cumbersome* for designers to understand;
  - implementation may be too *slow*;
  - spec and implementation can be *buggy*. 
Relationships between classes and objects

- **Association**: objects “related” but one does not own the other.
  
  ![Relationship Diagram](https://via.placeholder.com/150)

- **Aggregation**: complex object comprises several smaller objects.
  
  ![Aggregation Diagram](https://via.placeholder.com/150)

- **Composition**: *strong* aggregation: part may belong to only one whole – deleting whole deletes parts.
  
  ![Composition Diagram](https://via.placeholder.com/150)

- **Generalization**: define one class in terms of another. Derived class inherits properties.
  
  ![Generalization Diagram](https://via.placeholder.com/150)
Association Example

Keypad

| 1 | SendsNumberTo | 1 |

CellularRadio

Nature of the association

Optionally – show “direction” of association

SendsNumberTo
Aggregation/Composition Examples

Atoms may be in other lists
Deleting list doesn’t delete atoms.

Points can only be on one rectangle
Deleting rectangle deletes points.
Aggregation/Composition Examples

AddressBook

ContactGroup

Contact

n..m - between n and m instances
0..* - any number of instances (or none)
1..* - at least one instance
1 - exactly one instance
Generalization/Class derivation

- May define one class in terms of another (more “general”) class.
  - Instead of creating a new class
- Derived class inherits attributes & operations of base class.

\[
\text{Derived\_class} \quad \downarrow \quad \text{Base\_class}
\]

UML generalization
Class derivation example

Display
- pixels
- elements
- menu_items

parent class

pixel()
set_pixel()
mouse_click()
draw_box

base class

BW_display

child class

Color_display

child class

generalizations
Multiple inheritance

Speaker  
Display

Multimedia_display

derived class inherits properties of **both** base classes
Generalization example
Links and associations

- **Association**: describes relationship between *classes*.
  - Association & class = *abstract*

- **Link**: describes relationships between *objects*.
  - Link & object = *physical*
Association & link examples

Class Diagram

ADPCM: adaptive differential pulse-code modulation

Object Diagram

msg: ADPCM_stream
length : integer

Contains

0..*

# contained messages

# containing message sets

message set
count : integer

m1:message
msg = msg1
length = 1102

Msg:message set
count = 2

m2:message
msg = msg2
length = 2114

contains

contains

Contains

(association)

(association)

(association)

(association)
Object & Class Diagram Example

Object diagram

Class diagram
OO implementation in C++
(derive from UML diagram)

/* Define the Display class */
class Display {
    pixels : pixeltype[IMAX,JMAX]; /* attributes */

public:
    /* methods */
      Display(); { } /* create instance */
    pixeltype pixel(int i, int j) { 
        return pixels[i,j]; }
    void set_pixel(pixeltype val, int i, 
                   int j) { pixels[i,j] = val; }
};
Instantiating an object of a class in C++

/* instantiate Display object d1 */
Display d1;

/* manipulate object d1 */
apixel = d1.pixel(0,0);
d1.set_pixel(green,18,123);
Behavioral descriptions

- Several ways to describe behavior:
  - internal view;
  - external view.

- Dynamic models:
  - **State diagram**: state-dependent responses to events
  - **Sequence diagram**: message flow between objects over time
  - **Collaboration diagram**: relationships between objects

- Specify:
  - inter-module interactions
  - order of task executions
  - what can be done in parallel
  - alternate execution paths
  - when tasks active/inactive
State machines

Similar to sequential circuit state diagrams
Event-driven state machines

• Behavioral descriptions are written as event-driven state machines.
  • Machine changes state on occurrence of an “event”.
• An event may come from inside or outside of the system.
  • Signal: asynchronous event.
  • Call: synchronized communication.
  • Timer: activated by time.
• May also have state changes without events
  • Ex. when some condition is satisfied
Signal event

<<signal>>
mouse_click

leftorright: button
x, y: position

event declaration

mouse_click(x,y,button)

event description
Call event

draw_box(10,5,3,2,blue)

c \rightarrow d

Timer event

tm(time-value)

e \rightarrow f

Ex. RTOS “system tick timer”
Example: click on a display

start

mouse_click(x,y,button)/find_region(region) -> region found

region = menu/which_menu(i) -> got menu item

call_menu(I) -> called menu item

region = drawing/___find_object(objid) -> found object

highlight(objid) -> object highlighted

finish
Sequence diagram

- Shows sequence of operations over time.
  - Use to plan timing of operations
  - Relates behaviors of multiple objects.
- Objects listed at top from left to right
  - Each object has a time line (shown as dashed line)
  - Focus of control (shown as a rectangle) indicates when object is “active”
  - Actions between objects shown as horizontal lines/arrow
Sequence diagram example

Programs on a CPU: only one has control of CPU at a time

m: Main

f1: Function

f2: Function

box = “focus of control”

return(r1)

return(r2)
Sequence diagram example

Display and menu co-exist (both “active”)

m: Mouse

mouse_click(x,y,button)

which_menu(x,y,i)

call_menu(i)

lifelines

d1: Display

u: Menu

box = “focus of control”
Collaboration Diagram

- Show **relationship** between object in terms of messages passed between them
  - Objects as icons
  - Messages as arrows
  - Arrows labeled with **sequence numbers** to show order of events
Example: Cell phone class diagram

Cell phone **use case**: Make call

1. User enters number (presses buttons)
2. Update display with digits
3. Dialer generates tones for digits – emit from speaker
4. User presses “send”
5. “In use” indicator lights on display
6. Cell radio connects to network
7. Digits sent to network
8. Connection made to called party

Collaboration diagram: Make call

Show collaborations in the previous use case (including order)

1.1 DisplayDigit(code)
1.2 EmitTone(code)

2.1 Connect(pno)

Summary

- **Example: Model train set** (Section 1.4)

- Object-oriented design helps us organize a design.
- **UML** is a transportable system design language.
  - Provides structural and behavioral description primitives.