CS 351
Design of Large Programs
Object-Oriented Design Principles

September 6, 2021

A Starting Point

Simplifying assumptions:

- the program execution is sequential
- the program executes on a single machine

The program is hierarchically structured in terms of three levels:

- main program
- subordinate objects
- external devices

Relevant Concepts

Main program

- an *active procedure*
- controls the execution logic
- invokes methods on subordinate objects

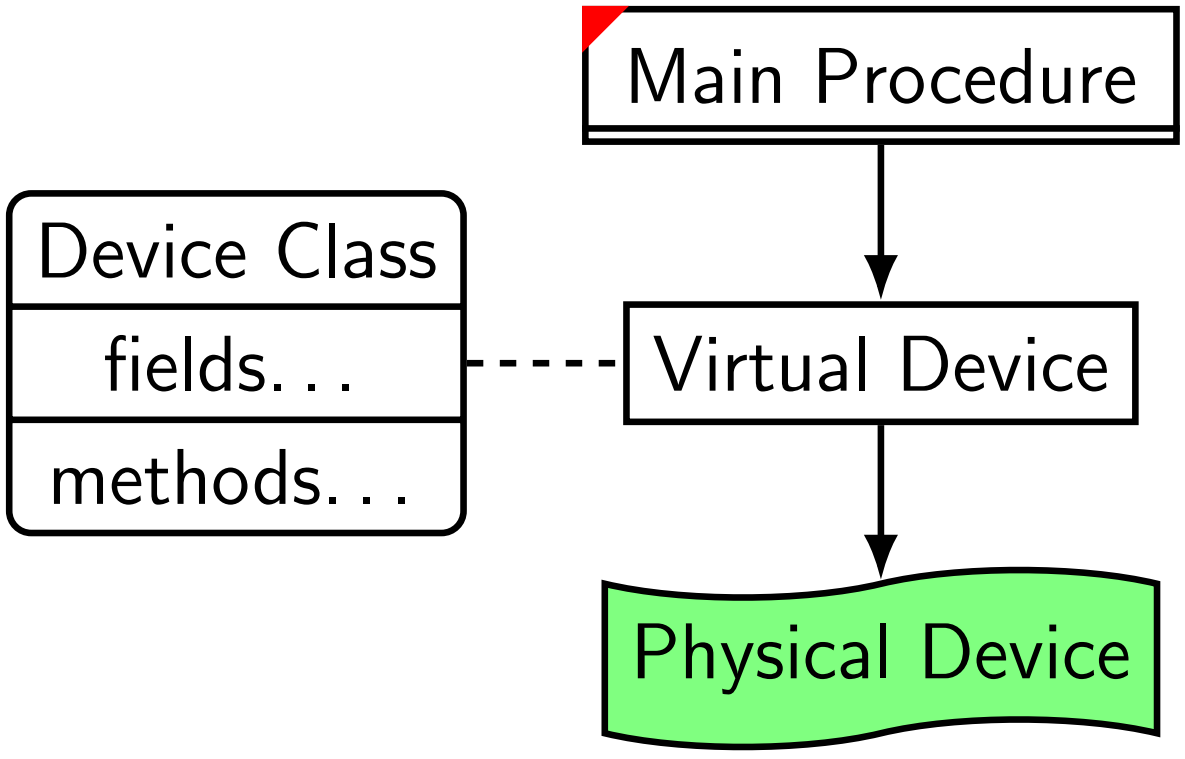
Subordinate objects

- are *objects* in the programming sense
- offer public methods to the main program
- do not interact with each other
- have no public fields

Key Relations

- The relation between the main program and the subordinate objects is a *reference* relation
  - the entity above may invoke services provided by the entity below i.e., the procedure may call methods on the objects below
- The relation between objects and external devices is an *encapsulation* relation
  - an external device is encapsulated by a single object
  - access to the external device below is controlled by the object above

Notation


Main Procedure
Device Class
fields...
methods...
Virtual Device
Physical Device

Design Principles

1. Separation of Concerns
2. Information Hiding
3. Data Encapsulation
4. Device Encapsulation
5. Balanced Levels of Abstraction
6. Protection Against Change

1. Separation of Concerns

- The principle of separation of concerns demands that:
  - unrelated concerns should be associated with distinct entities in the design
  - related concerns should be associated with a relevant entity in the design
- This principle impacts design decisions relating to modularity
- Object-oriented design enables the application of this principle
- Strict application of the principle is not always straightforward
- Changes to requirements may have a major impact on the design

Example: HTML, CSS, and Javascript

- HTML: HyperText Markup Language, used to specify the contents of a webpage
- CSS: Cascading Style Sheets, used to specify the style of a webpage
- Javascript, used to specify the behavior of the webpage
- Each piece has it own separate use and a change in one should* not affect the others

2. Information Hiding

Limit knowledge about design decisions as much as possible.

- fundamental to encapsulation

Postpone design decisions for as long as possible.

- fundamental to top-down design

This relates strongly to the scope of program changes. . . How we can minimize them?

3. Data Encapsulation

- Bundling data with methods that operate on this data
- Decoupled implementation details from operations on the data.
- This simplifies programming.

Illustration: Custom Dictionary

Consider an object called MyDictionary:

- initially contains an empty set W of words
- at most N words can be stored
- addWord(w) – adds one word to the set W, if there is room for it
- removeWord(w) – removes one word from the set W
- containsWord(w) — returns true iff the word is in the set W

Simple Implementation: array of strings

Illustration: Custom Dictionary Revisited

Consider the following change in requirements:

- ~~at most N words can be stored~~
- the number of words is very large

This new implementation requires a tree structure.

4. Device Encapsulation

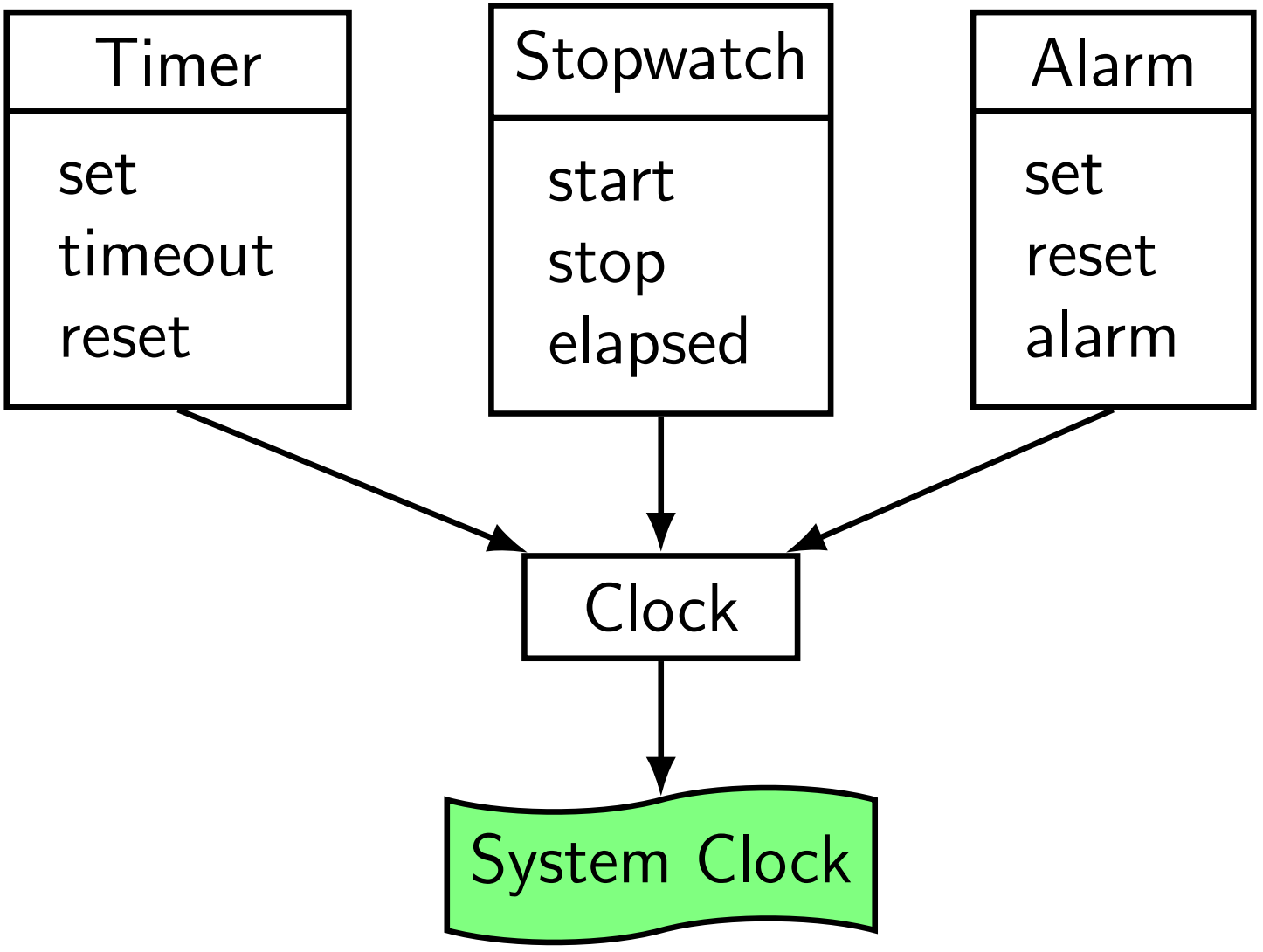
Devices are a volatile element of most designs. Protect the system against device/protocol substitutions:

- microcontroller reassignment of pins
- communication interface (USB connection vs. Ethernet)
- memory mapped I/O

Layers of encapsulation:

- application-specific virtualization
- virtual device
- device driver

Illustration: Timers

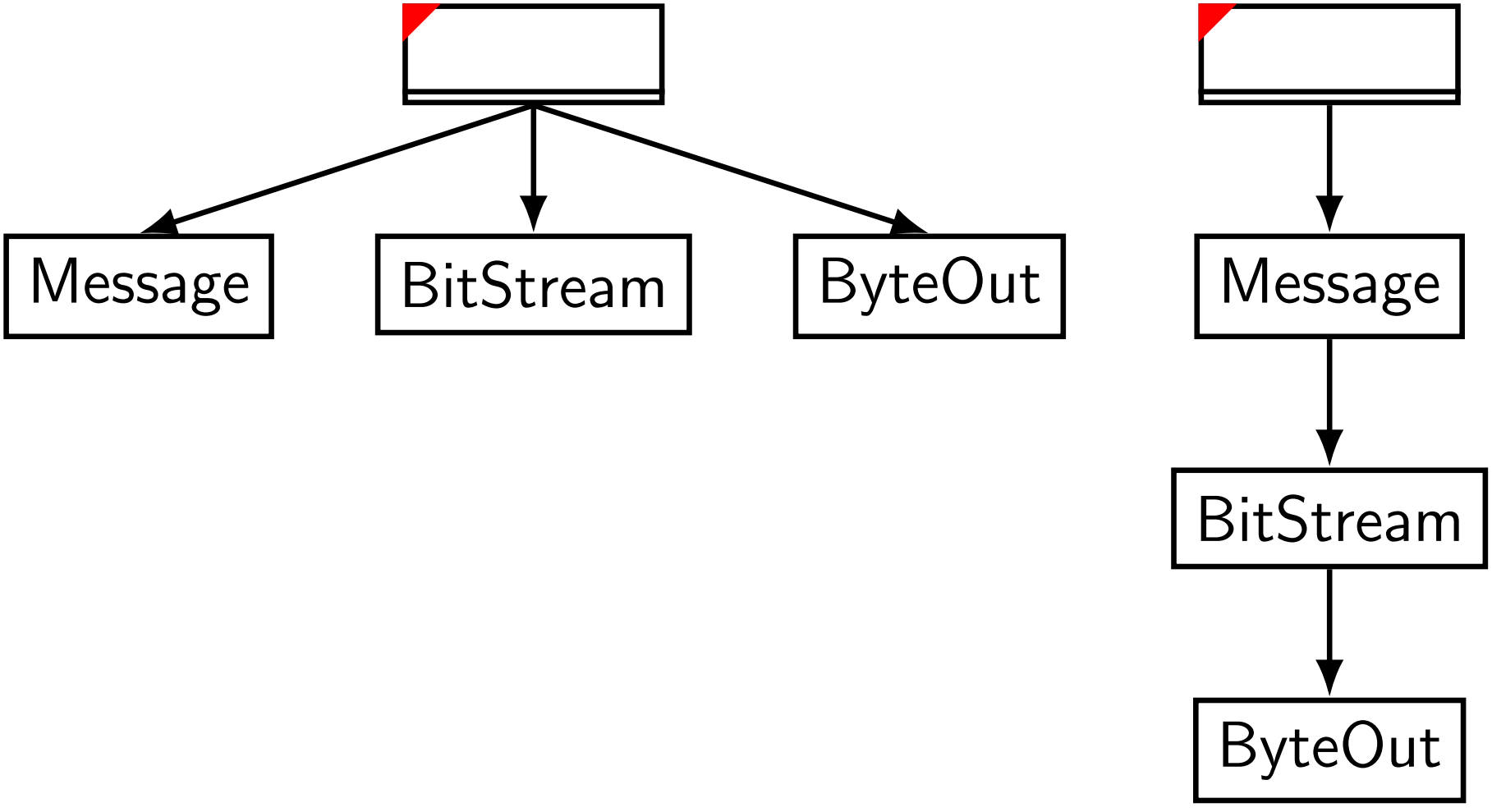


5. Balanced Levels of Abstraction

In a hierarchically designed program or system:

- when moving up in the structure the level of abstraction should increase
- when moving down in the structure the level of abstraction should decrease
- entities at the same level in the structure should exhibit comparable degree of abstraction

Illustration: Message Delivery



6 Protection Against Change

The fundamental engineering concern of object-oriented design is *to protect the design and implementation against impact of potential changes*

- modifications to delivered code are expensive
- modifications can introduce errors
- limiting the scope of potential changes reduces cost and mitigates risks

Any proposed design needs to be analyzed with respect to the impact of changes

- processing logic
- processor changes
- device substitution
- elimination of performance bottle necks