Programming Language Design

2015

Week #5: mixins&trains / design patterns / frameworks

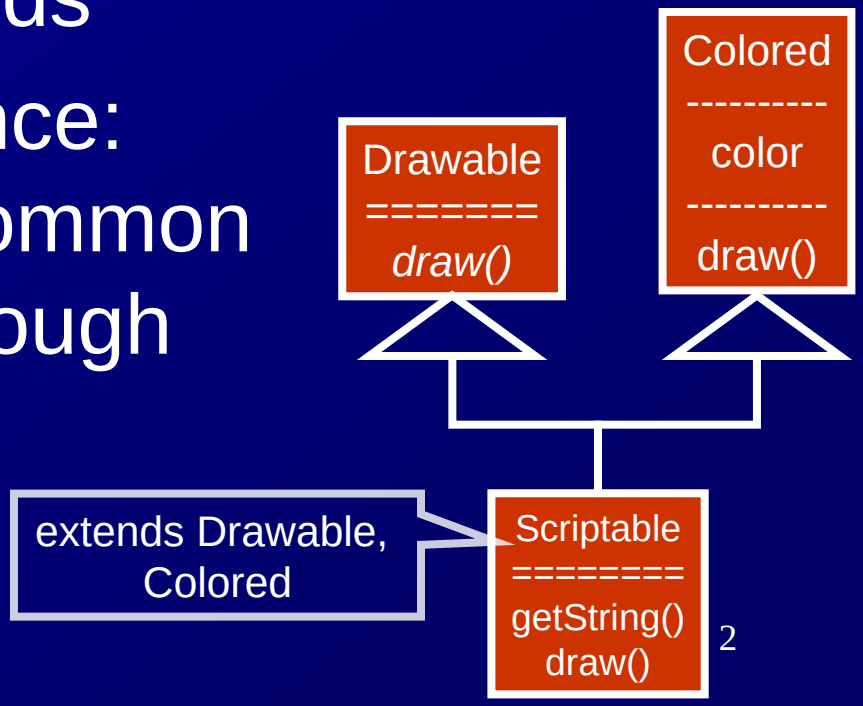
Instructor: Hidehiko Masuhara

(review) Pros&cons of multiple inheritance

pro) can reuse ***additional*** functions

con) ambiguity: when parents define the same name methods

con) diamond inheritance: where there is a common ancestor found through different parents



Traits and mixins

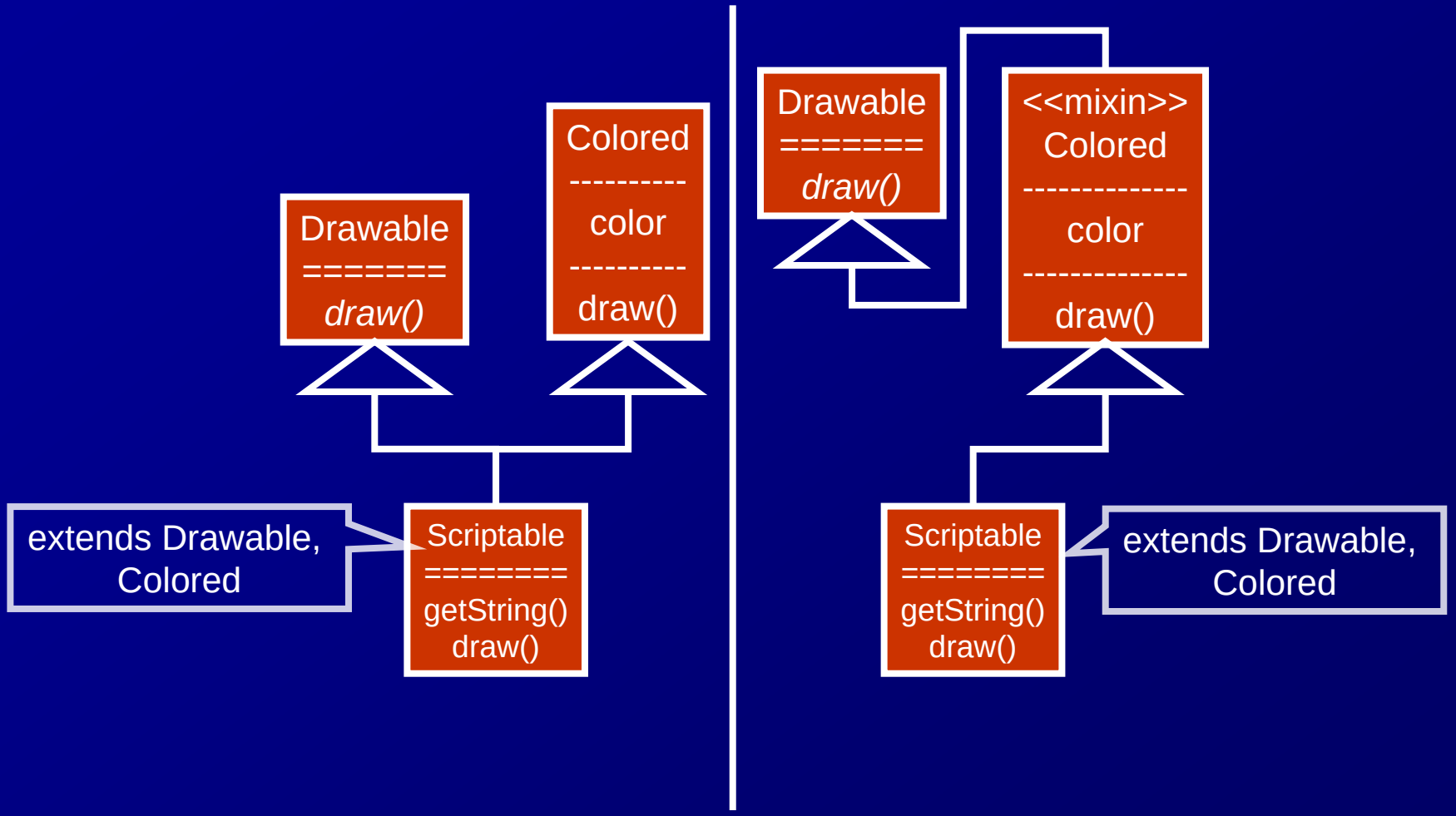
- Observation: many use cases of multiple inheritance is to reuse ***additional*** functions
- Let's provide a language feature for those cases
  - while keeping single inheritance (ie, only one parent)
- Two solutions:
  - mixins [BC90] — by linearization
  - traits [SDNP03] — by flattening

  (Confused naming: traits in Scala are mixins)
- (Other criteria: whether it can have instance variables)

mixins [BC90]

- A mixin M is a class whose superclass can be specified later
- C inherits M and class S, M is inserted in betwen C and S (linearization)
  - no ambigoutiy: mixins always come first
  - no diamond inheritance: each use of mixin is distinct
- A mixin can inherit from anthor mixin
- We cannot create an object only from a mixin
- Example: ***traits*** in Scala

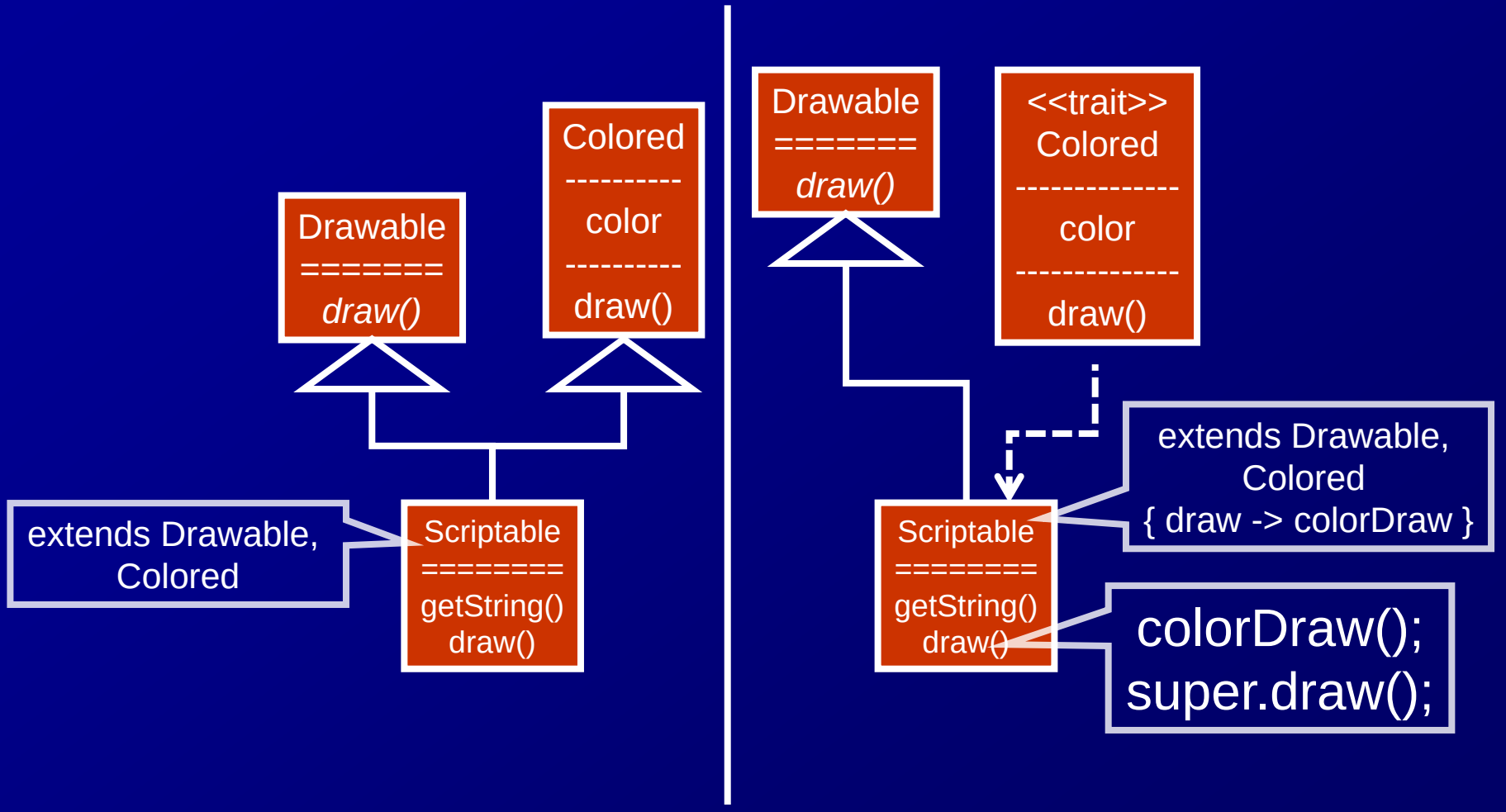
Multiple inheritance vs mixins



Traits [SDNP03]

- Traits are very similar to mixins
- Difference
  - When C inherits trait T and class S, all the definitions in T are ***copied*** into C
  - When definitions have the same name, names of those in T must be renamed (flattening)
    - no ambiguity / no diamond inheritance

Multiple inheritance vs. traits



Quiz 1/2 (10min.)

- Design a 3D CAD program
- Casts
  - Volume — each thing in the 3D space
  - View — front and side views
- Use cases
  - (When the user drags a volume, it calls move() on the volume object.) Move() changes the position of the volume
  - (When the animation command is executed, the command calls the move() method on volue objects.)
  - When a position of a volume is changed, draw() is called on the front and side views. (Draw() shows an updated scene)
- Note: only design underlined parts

Quiz 2/2 (10min.)

Provide common functions for game programs as a component

- common funcs = outside of yellow boxes
- in any language (OOP)
- also show how to use the component to implement a game

```
byte[] imageBuf1 = ..., iageBuf2 = ...;
int previousKey = 0;
while (true) {
    int c = read keyboard status;
    int key = determine "currently pressed
                key" from c and previousKey;
    previousKey=key;
    switch (key) {
    case Left: move to left; break;
    case Right: move to right; break;
              ...;
    }
    move enemies;
    clear(imageBuf1);
    draw charactrs on imageBuf1;
    transferToDisplay(imageBuf1);
    swap imageBuf1 and imageBuf2;
    wait for a while;
}
```

Design Patterns

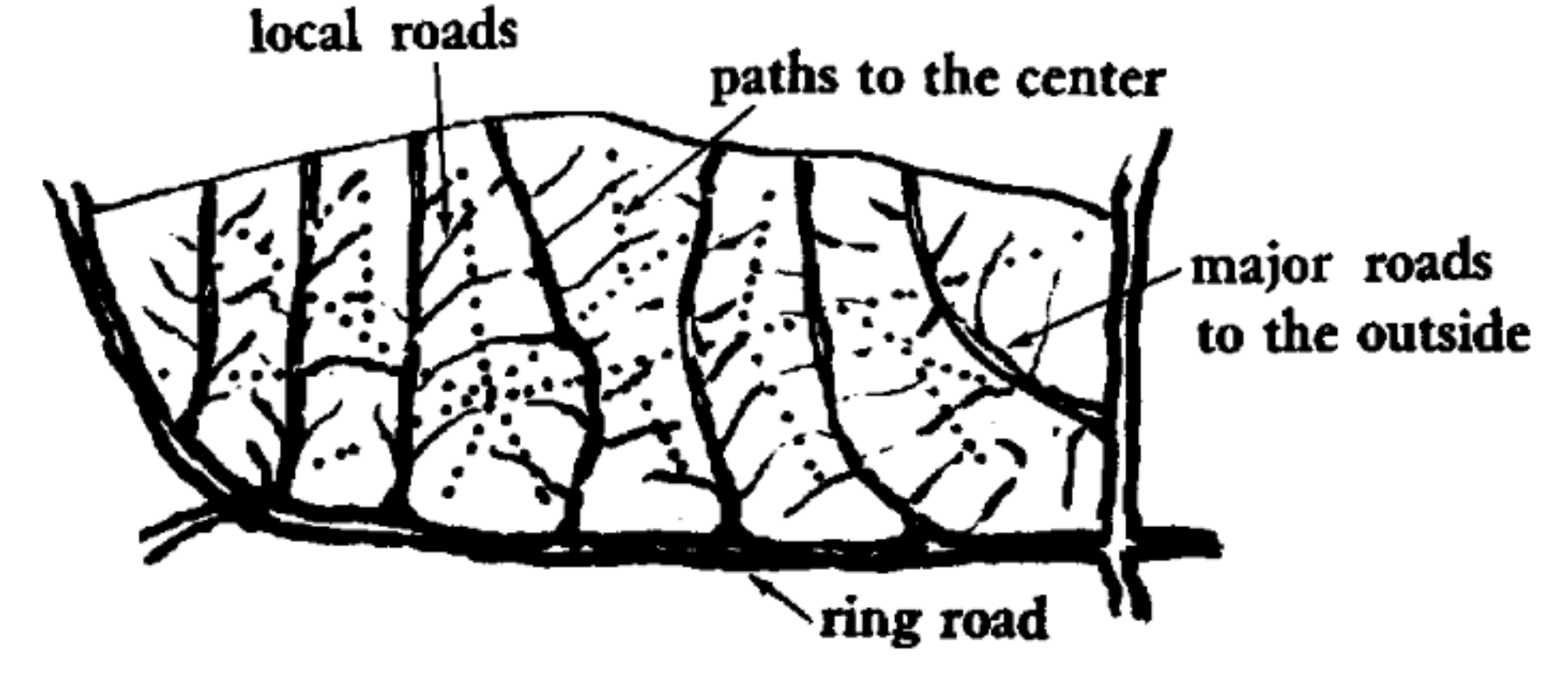
Design Patterns for Reuse: the Origin

Alexander's design patterns [Alexander77]

- Domain: designs of inside of a building to an entire city
- Scale: various (depends on the domain)
- Target: interactions between a few elements (eg: stairs and doors)
- Purpose: a language for describing interactions between elements

NB: not a catalog of components

Example of a pattern in the Alexander's book



- layout of several types of roads in a city district

Design patterns for reuse: the Gang of Four (GoF) Patterns [GoF94]

Domain: class design

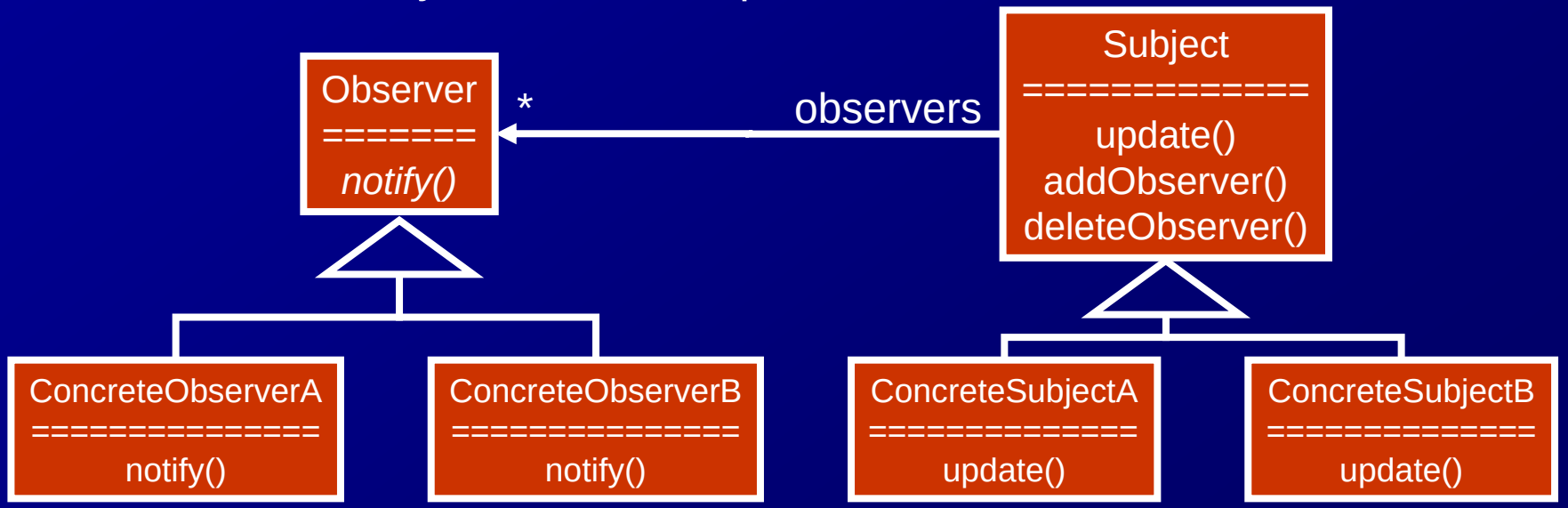
Scale: a few classes

Purpose: making a class design is more "reusable"

- a change in a specification can be realized without modifying existing classes
- does not mean "reusing" design patterns per se

An example design pattern: Subject-Observer

- Name: Observer Pattern
- Purpose: define one-to-many relationship between objects, and when a state of an object is changed, all the dependent objects are automatically notified and updates their states

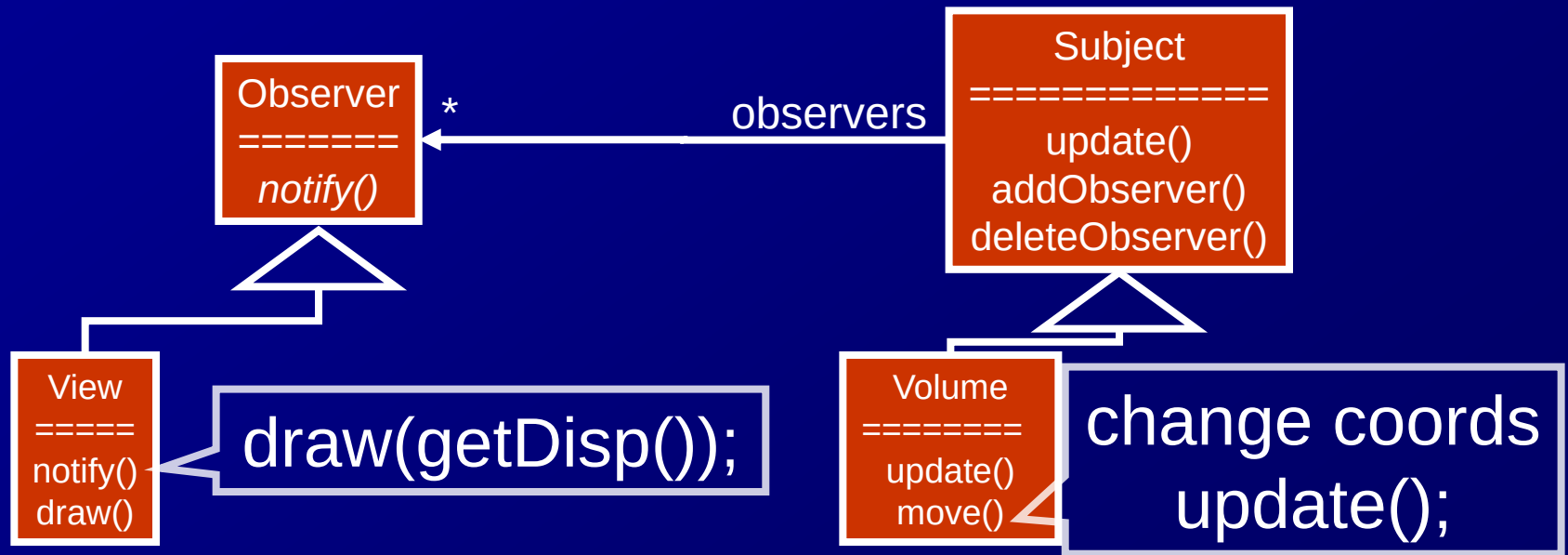


Quiz 1/2 (10min.)

- Design a 3D CAD program
- Casts
  - Volume — each thing in the 3D space
  - View — front and side views
- Use cases
  - (When the user drags a volume, it calls move() on the volume object.) Move() changes the position of the volume
  - (When the animation command is executed, the command calls the move() method on volue objects.)
  - When a position of a volume is changed, draw() is called on the front and side views. (Draw() shows an updated scene)
- Note: only design underlined parts

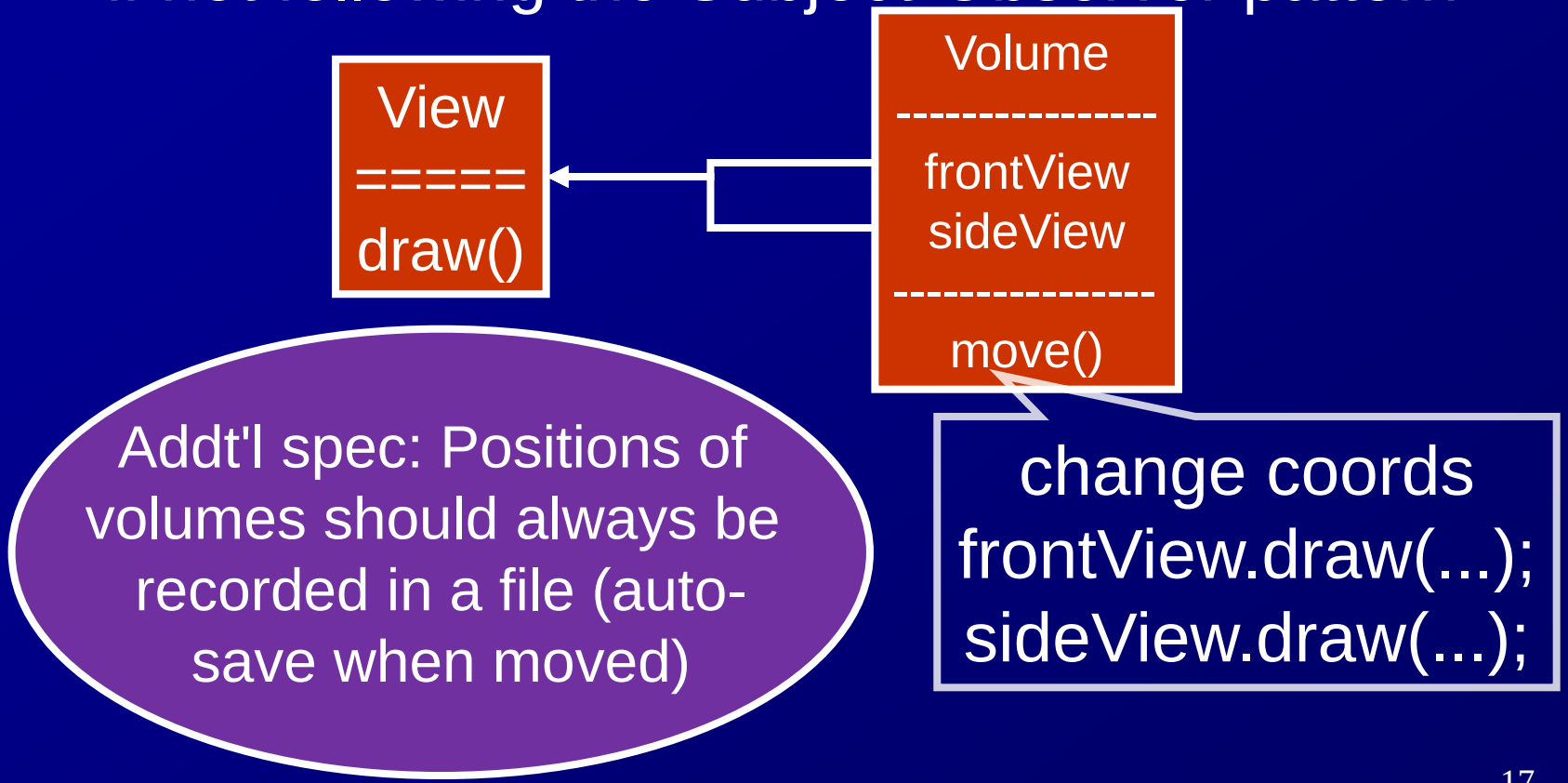
An application of a design pattern: Subject-Observer

- 3D CAD program
- It shows volumes in a space from two views, namely front and side. When a volume moves, all the views are refreshed



How design patterns make classes more reusable?

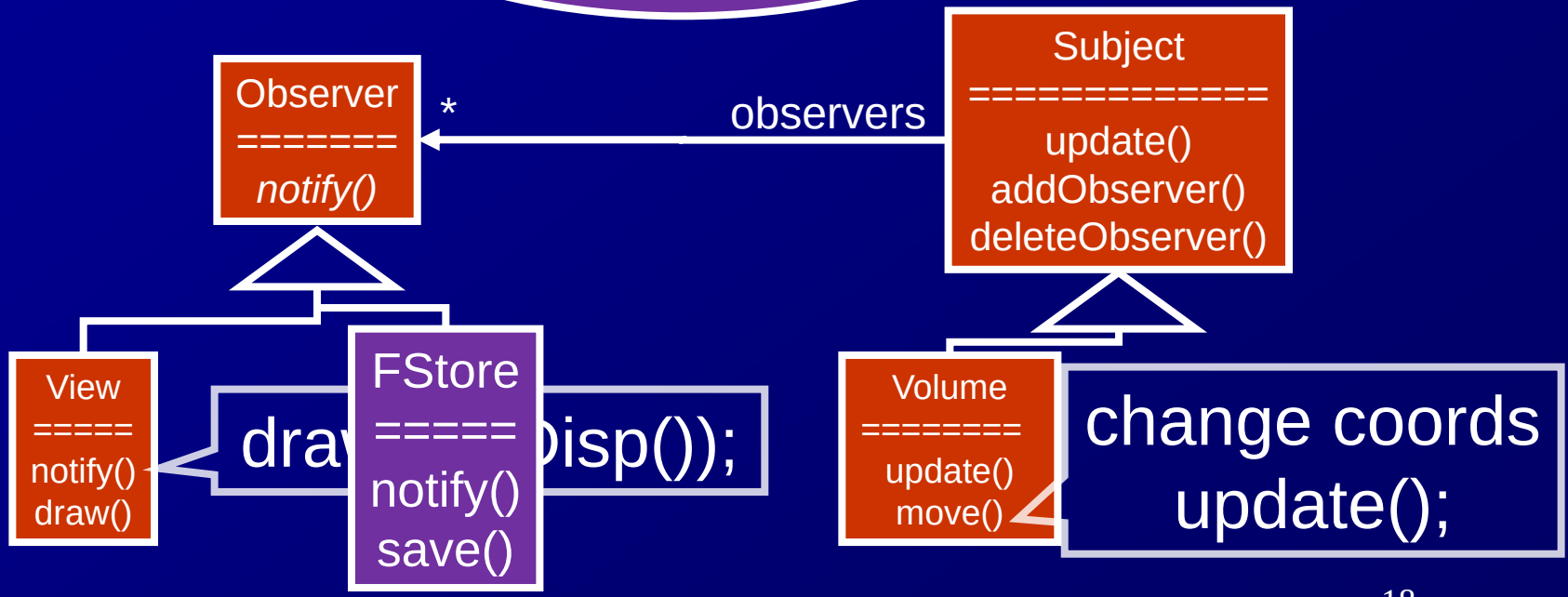
- if not following the Subject-Observer pattern



An application of a design patter server

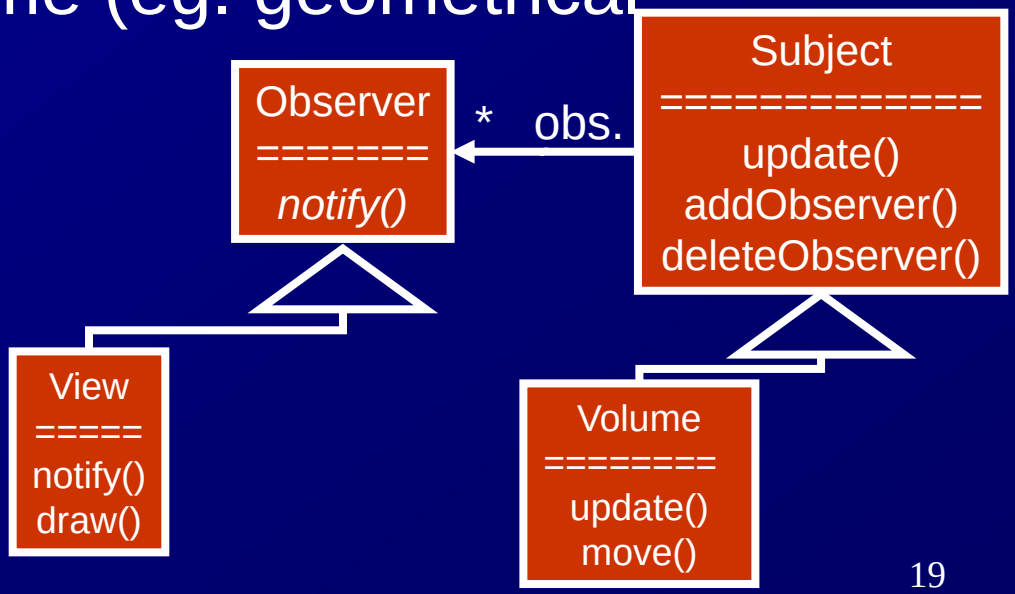
- 3D CAD pro
- It shows volu ly front and side. When a v efreshed

Addt'l spec: Positions of volumes should always be recorded in a file (auto-save when moved)

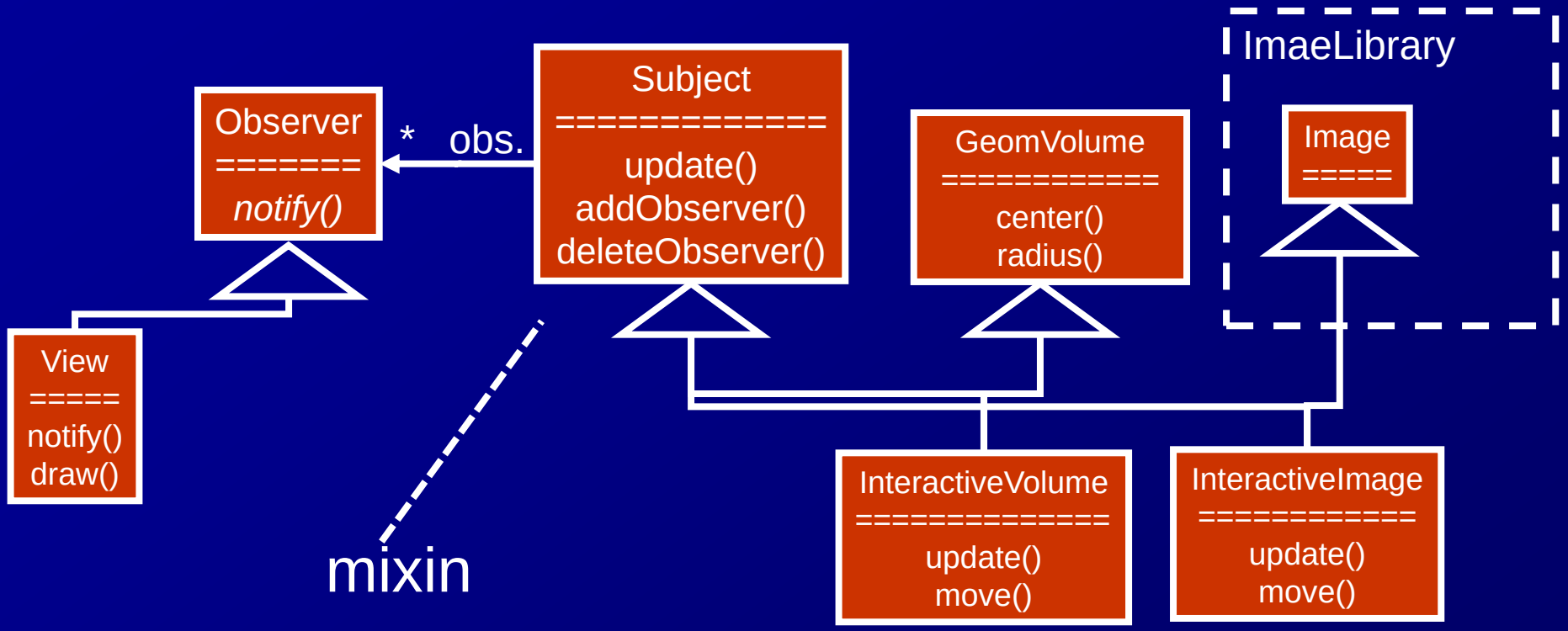


Subject-observer pattern with traits/mixins

- Subject offers functions to add / delete / update observers
- We want to have a specific class as a superclass of Volume (eg. geometrical volume with no displaying funcs)
- We want to show existing Image class



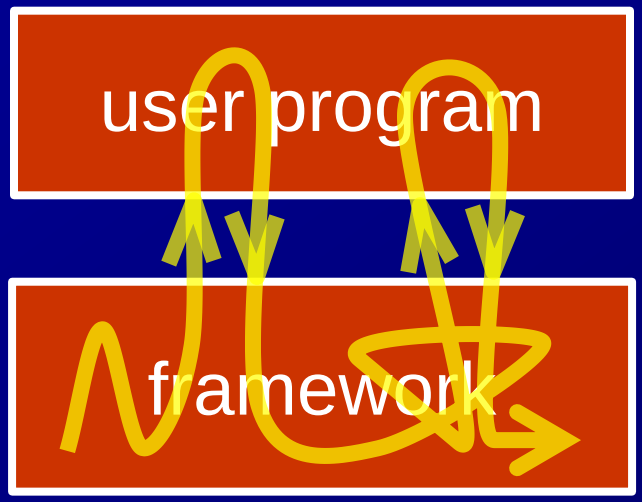
Subject-observer pattern with traits/mixins

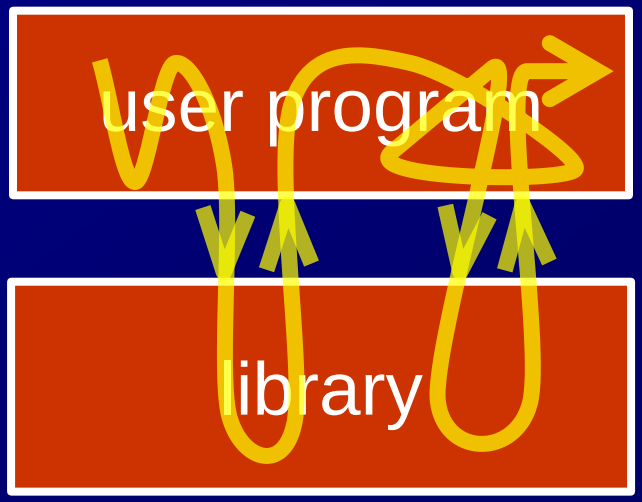


Frameworks

Frameworks vs Libraries: different styles of program reuse

- Framework: reuse control flow as well (active)
- Library (narrow): passively provide services





How frameworks are provided

- Procedural / functional languages: the user defines callback functions and provides them to a framework's function (eg. GUI framework on X11 window system)
- OOP languages: a framework provides a "base" class. The user creates a subclass and fill the "holes" (methods that perform specific behaviors) by overriding
  - The base class can also define default behaviors (by not overriding)

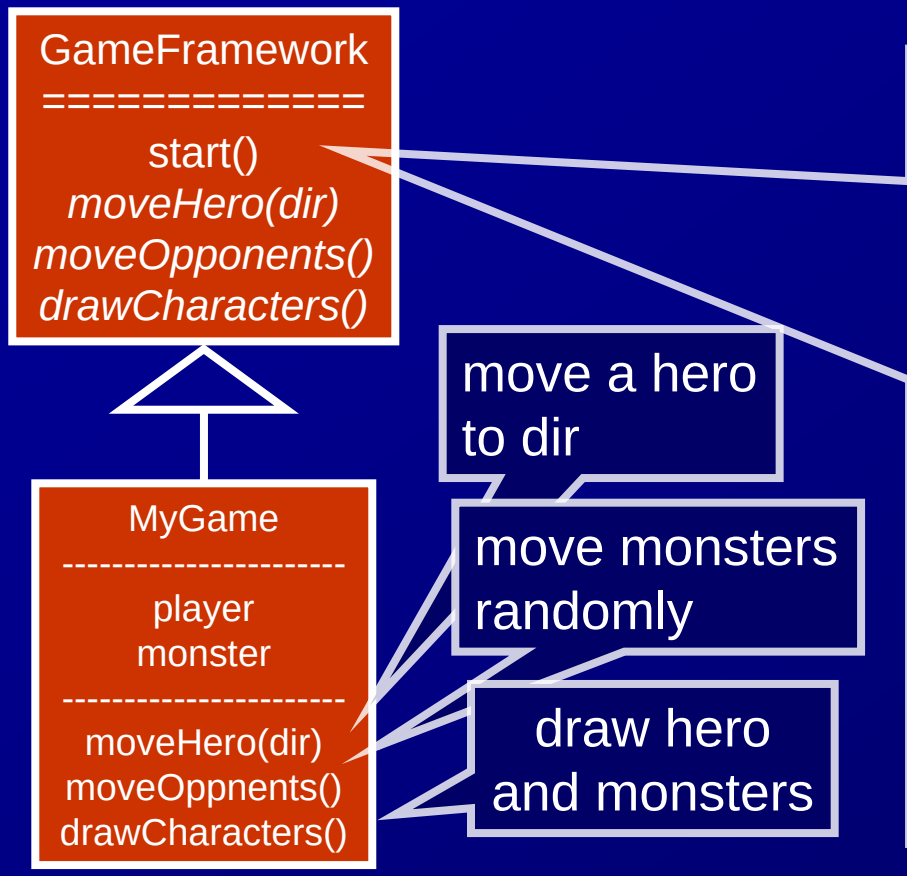
Quiz 2/2 (10min.)

Provide common functions for game programs as a component

- common funcs = outside of yellow boxes
- in any language (OOP)
- also show how to use the component to implement a game

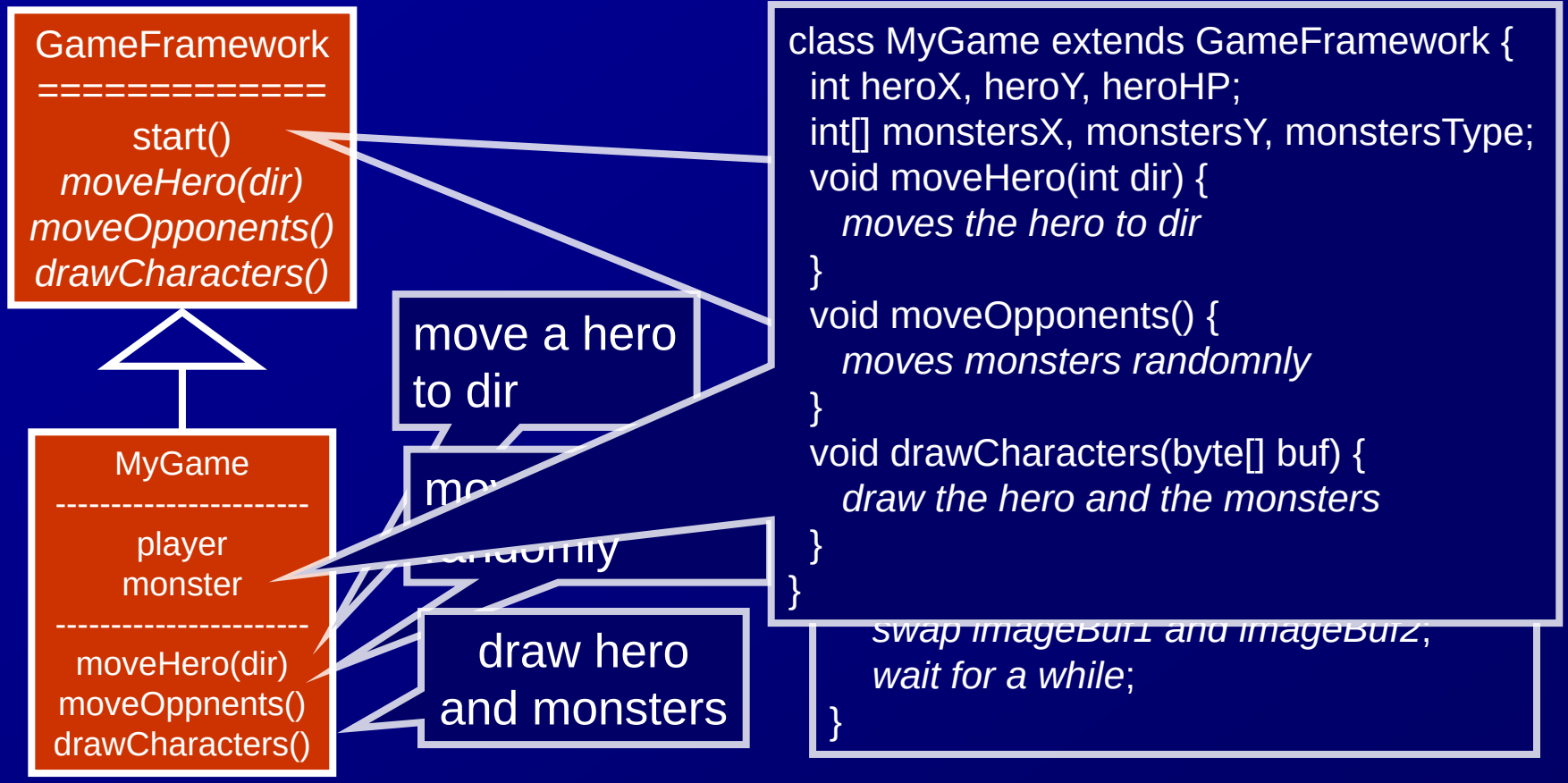
```
byte[] imageBuf1 = ..., iageBuf2 = ...;
int previousKey = 0;
while (true) {
    int c = read keyboard status;
    int key = determine "currently pressed
                key" from c and previousKey;
    previousKey=key;
    switch (key) {
    case Left: move to left; break;
    case Right: move to right; break;
              ...;
    }
    move enemies;
    clear(imageBuf1);
    draw charactrs on imageBuf1;
    transferToDisplay(imageBuf1);
    swap imageBuf1 and imageBuf2;
    wait for a while;
}
```

An example of a framework




```
byte[] imageBuf1 = ..., iageBuf2 = ...;
int previousKey = 0;
while (true) {
    int c = read keyboard status;
    int key = determine "currently pressed
                key" by c and previousKey;
    previousKey=key;
    this.moveHero(key);
    this.moveOpponents();
    clear(imageBuf1);
    drawCharacters(imageBuf1);
    transferToDisplay(imageBuf1);
    swap imageBuf1 and imageBuf2;
    wait for a while;
}
```

An example of a framework



OOP-based framework, from a historical perspective

```
main(argc, argv)
int argc;
char **argv;
{
     Widget topLevel, goodbye;

     topLevel = XtInitialize(
          argv[0],
          "XGoodbye",
          NULL,
          0,
          &argc,
          argv
          );

     goodbye = XtCreateManagedWidget(
          "goodbye",                  /* arbitrary widget name */
          commandWidgetClass,         /* widget class from Command.h */
          topLevel,                   /* parent widget*/
          NULL,                       /* argument list */
          0                           /* arg list size */
          );

     XtAddCallback(goodbye, XtNcallback, Quit, 0);

     /*
      *  Create windows for widgets and map them.
      */
     XtRealizeWidget(topLevel);

     /*
      *  Loop for events.
      */
     XtMainLoop();
}
```

objects

```
/*
 * Quit button callback function
 */
/* ARGSUSED */
void Quit(w, client_data, call_data)
Widget w;
caddr_t client_data, call_data;
{
     fprintf(stderr, "It was nice knowing you.\n");
     exit(0);
}
```

call back function

register a call-back function

framework's main

Nye and O'Reilly, *X Toolkit Intrinsics Programming Manual*, O'Reilly & Associates, 1990, p.40

References

[SDNP03] Schärli, Nathanael, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. "Traits: Composable units of behaviour." In ECOOP 2003–Object-Oriented Programming, pp. 248-274, 2003.

[BC90] Bracha, Gilad, and William Cook. "Mixin-based inheritance." In *Proceedings of OOPSLA/ECOOP,* pp.303-311, 1990.

[Alexander77] Alexander, Christopher, A Pattern Language: Towns, Buildings, Construction. Oxford University Press, 1977

[GoF94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994