Design Patterns

Note Title

- Captures common designs & gives them names

  ⇒ Efficient design

- Eventually, languages may support patterns more directly

```
1:
    ⋮
    if — Goto 6
    Goto 1
6:
```

} ⇒ while loop

Factory Pattern

Problem: JList or JCombobox of Actions
choose an action for a given step

Solution A: Put Strings in list / combo box

```
get selection
   if (selection.equals("New Screen"))
      return new NewScreenAction()
   else if (selection.equals("Speak phrase"))
      return new SpeakPhraseAction();
   ⋮
```

return
selection.createAction()

Better: Use Factory pattern
Put in list a bunch of Factories
- createAction()
- toString would describe the action type

Variation: class is its own factory ...
Prototype instances

Flyweight Pattern

Problem:
- View of some collection
- Update view whenever membership changed
- DON'T want to create a new view for every object on every update

Solution:
- HashMap: Model Objects → View Objects
- when collection changes

```
[ for (Element e : collection)
      if (!map.containsKey(e))
          map.put(e, factory.createView(e));
  add view for e to collection view from map
```

Weak HashMap

ViewMap — factory

<Element, View>

```
@override
public View get(Element e) {
    if (!containsKey(e))
        put(e, factory.createView(e));
    return super.get(e);
}
```

Singleton Pattern

Problem:
- Want a single audio Player object
- Want lots of objects to be able to refer to it, but ...
  - don't want to pass a reference everywhere
  - don't want to expose a static variable in a global way

```
public static final Player PLAYER = new Player();
```

OK, but may want to initialize on demand

Solution:

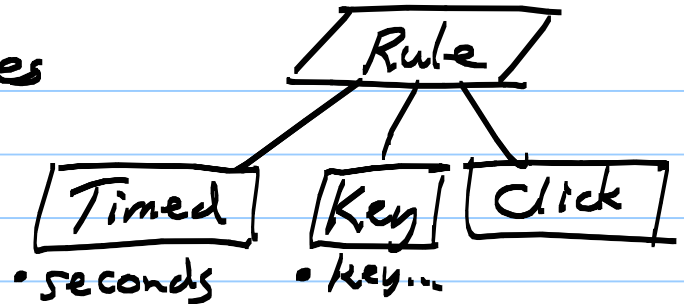
1. make the Player's constructor private
2. create a static method

```
public static Player getDefaultPlayer() {
    if (thePlayer == null)
        thePlayer = new Player();
    return thePlayer;
```

3. `private static Player thePlayer = null;`

State Pattern

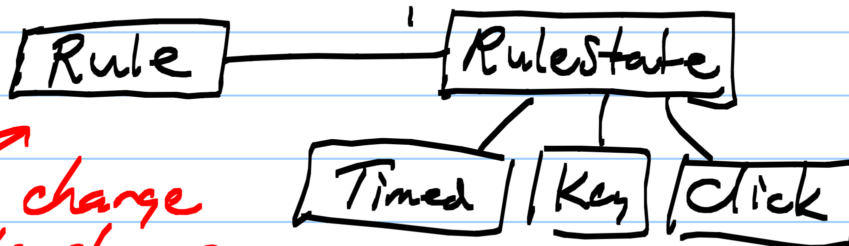
Problem:
- Have a hierarchy of Rule types
- Have a list of rules
- User can:
  - add a rule
  - change ~~the~~ rule type ← think about as operation on the rule



Options:

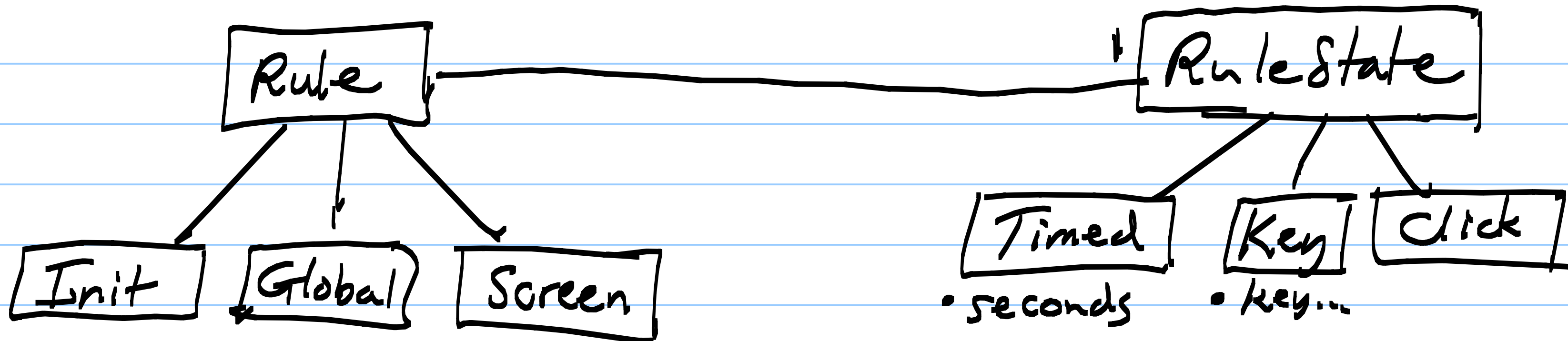
(A) Remove old rule from list & put a new rule in its place

(B) Use State Pattern

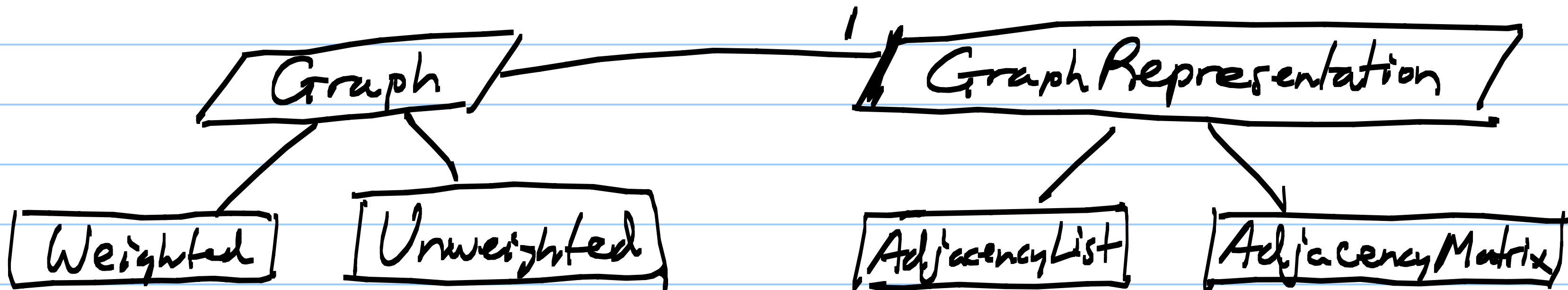


Can fire a change when its state changes

Bridge Pattern



Want to merge these two reasonable rule hierarchies



Command Pattern

Problem: Send an object over the network & do something when message arrives

```
if (message instanceof RemoveRule)
    ((RemoveRule) message).getRule().remove()
else if (message instanceof AddScreen)
    ---
else --
  ⋮
```

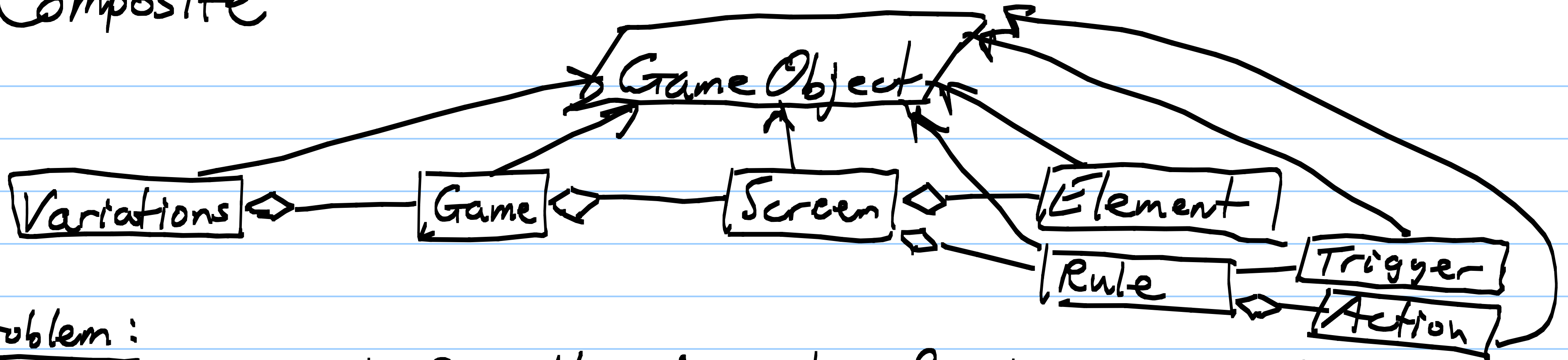
Solution: (A) Let all messages implement Runnable

```
message.run();
```

(B) Let messages implement a Command interface you define

```
public void execute(Game g, ...)
```

Composite

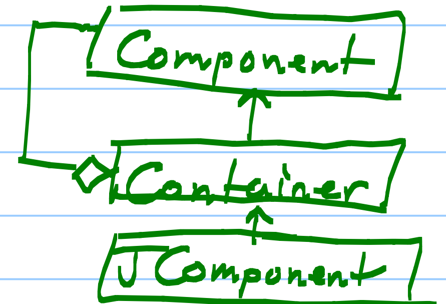


Problem:

Create a nicely formatted description of all game variations

⇒ traverse all these collection

Options:



① Iterate externally

```
for (Game g : vars)
    g....
    for (Screen s : g)
        ....
        for (Element e : s)
```

② procedurally, delegate at level of the composite...

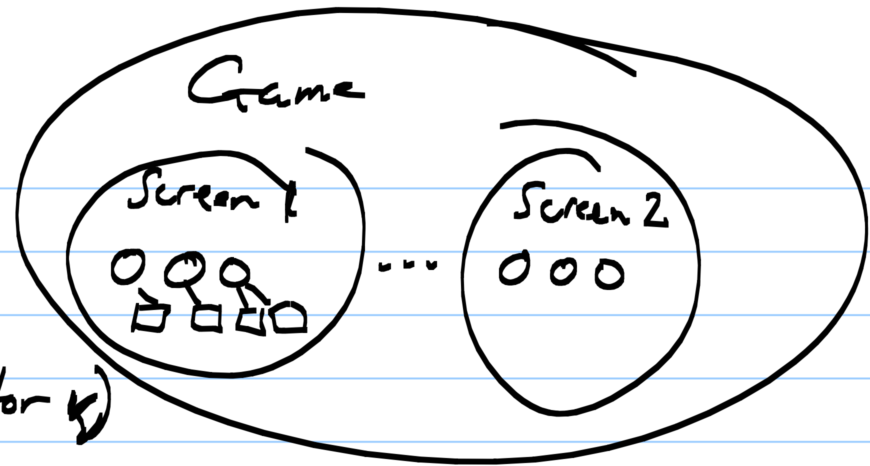
GameObject displayAsText(PrintStream p)

In Game:

```
p.println(name + ":\n");
for (Screen s : screens)
    s.displayAsText(p)
```

Advantage: No "omniscient" entity —
each object formats itself &
knows about its own collections

③ Visitor



GameObject accept(Visitor v)

```
interface Visitor {
    public void visit(Object);
}
```

Procedurally let visitor visit each object

Example: In Game class

```
public void accept(Visitor v) {
    v.visit(this);
    for (Screen s : screens)
        s.accept(v);
```

Additional design patterns we've seen:

- publish/subscribe
- model/view/controller
- iterator
- wrapper — i/o package
- plug-in — layout managers
- template pattern — abstract class w/ abstract methods
  "real" methods call abstract ones
- adaptor pattern
- proxy pattern — Java RMI