Design Principles

chiefly but not exclusively object-oriented

Objects, classes, and modules interact and change.

Decisions made to solve problems (e.g. design patterns).

Following good principles can isolate from change.

Key ideas:

Protected Variation [Larman]

Dependence on stability [Martin]

Change Control Principles

at the level of detailed design

Open-Closed Principle

Be open for extension; closed for modification.

Dependency Inversion Principle

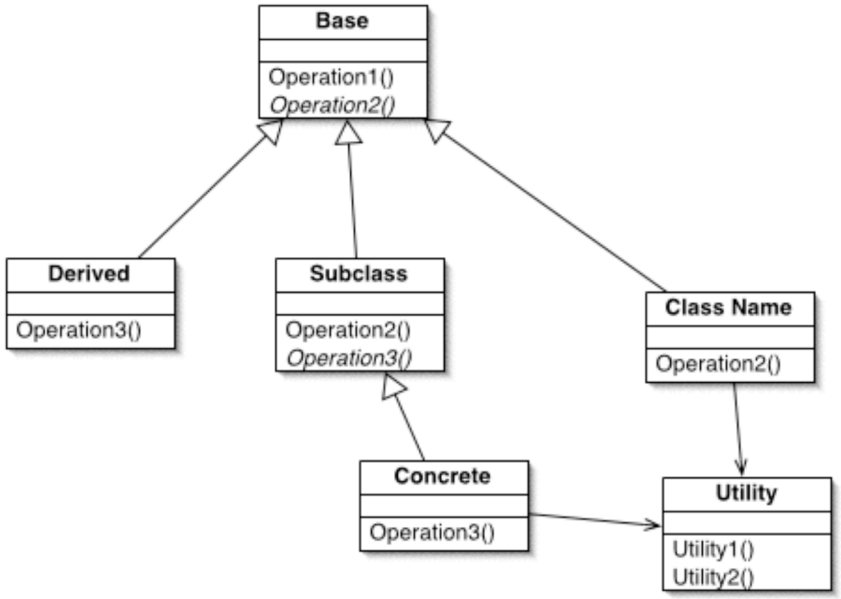
Depend on the more abstract.

Interface Segregation Principle

Implement (depend on) clients' interfaces.

Definitions

A class or module may be in itself more or less
concrete or *abstract*
and in context, more or less
stable or *volatile*
responsible or *irresponsible*
dependent or independent



Stability

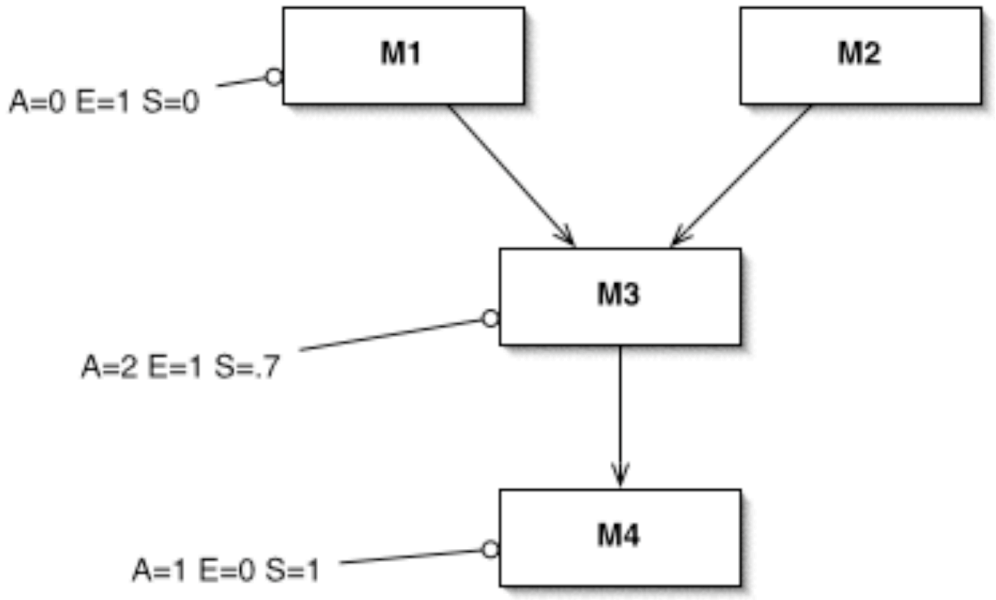
Measure of

how likely a module is to change
how frequently a module changes

One way to measure it: consider forces for change

afferent coupling = coupling into (*ad-*) this module
efferent coupling = couping out of (*ex-*) this module

$$\text{stability}(M) = \frac{A_M}{A_M + E_M}$$


M1
M2
A=0 E=1 S=0
M3
A=2 E=1 S=.7
M4
A=1 E=0 S=1

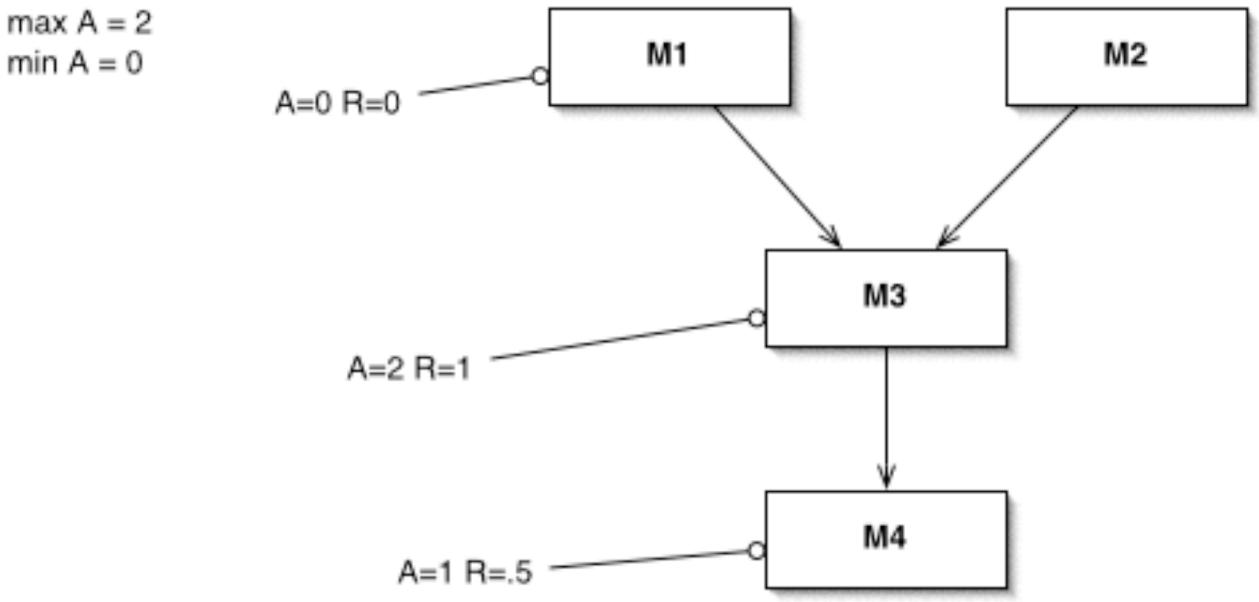
Responsibility

A module or class is more *responsible* when more other classes *depend* on it.

Dependence: calls, #includes, extends, implements, ...

One way to measure it: consider total afferents relative to maximum and minimum of the system:

$$\text{responsibility}(M) = \frac{A_M - A_{\min}}{A_{\max} - A_{\min}}$$


max A = 2
min A = 0
M1
M2
A=0 R=0
M3
A=2 R=1
M4
A=1 R=.5

Abstractness

A class or module is *abstract* when it
- has pure virtual functions (C++, C#, Java)
- is an «interface» (Java, C#)
- defines a prototype with no implementation (C, C++)
- depends on no other class or module (any language)

Hard to quantify at the level of individual modules.

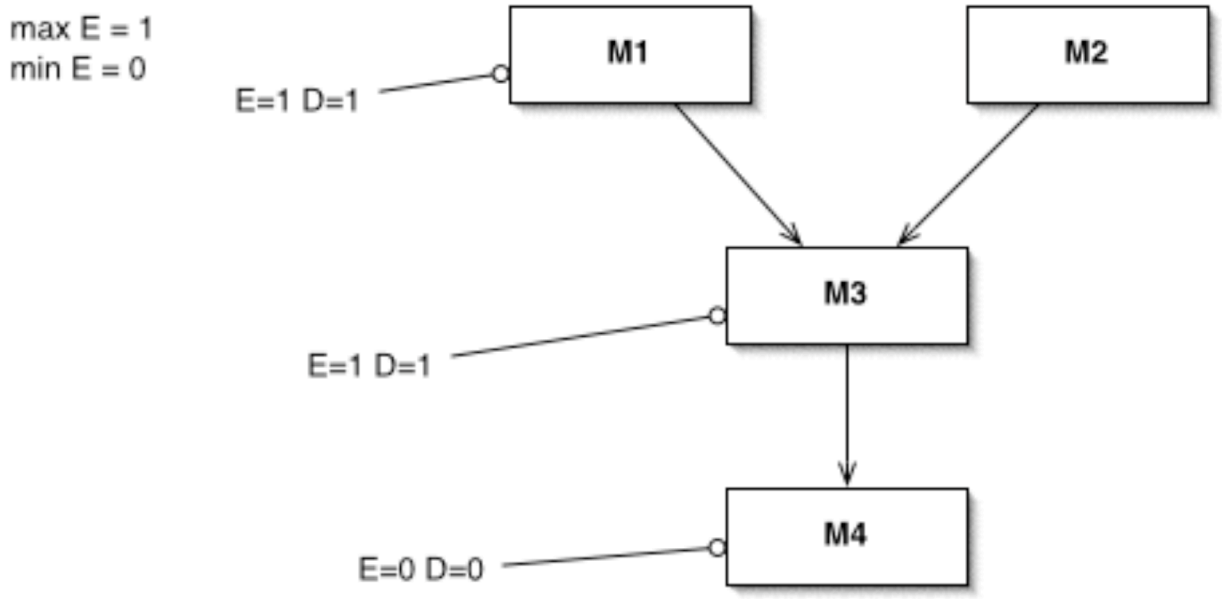
(…stay tuned)

Dependency

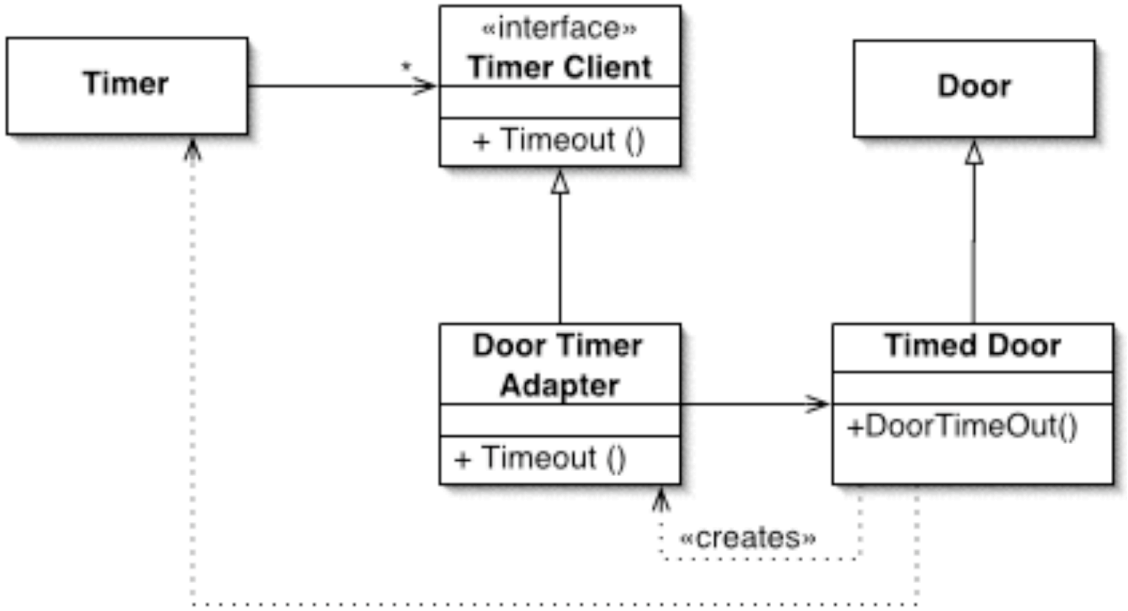
A module or class is more *dependent* when it depends more on other classes.

Possible metric: consider total efferents relative to maximum and minimum of the system:

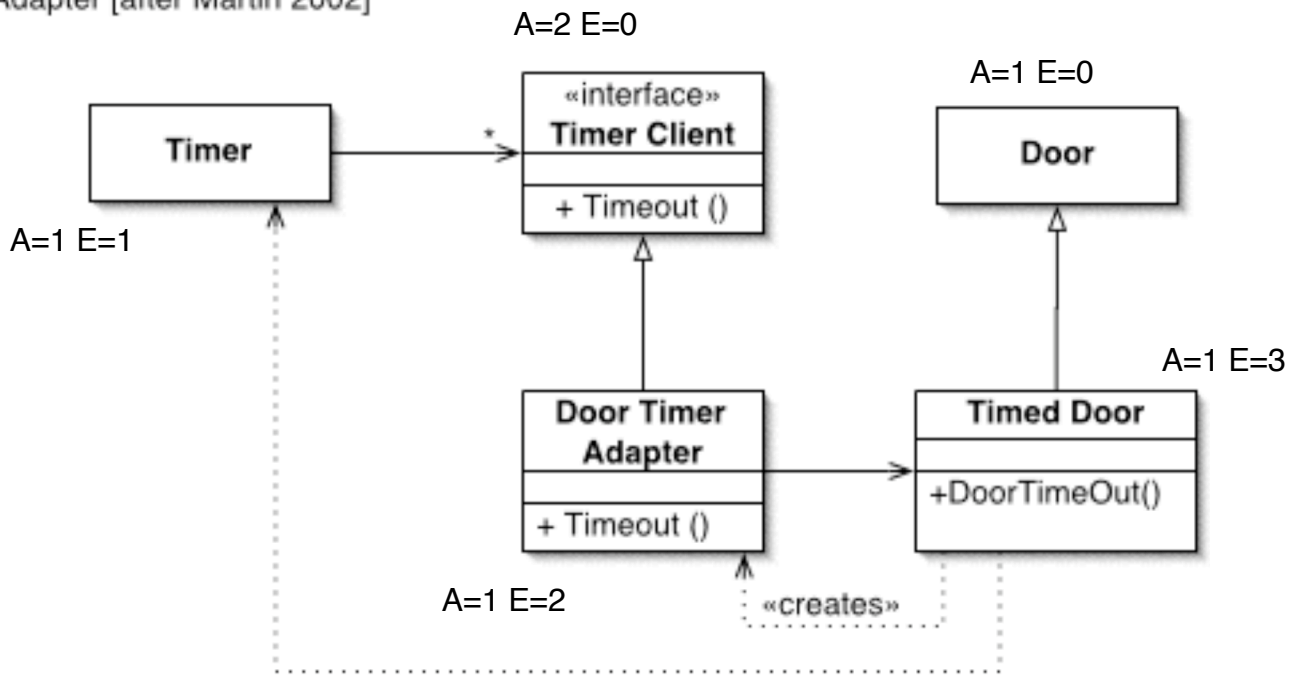
$$\text{dependency}(M) = \frac{E_M - E_{\min}}{E_{\max} - E_{\min}}$$


max E = 1
min E = 0
E=1 D=1
M1
M2
M3
E=1 D=1
M4
E=0 D=0

Door Timer Adapter [after Martin 2002]

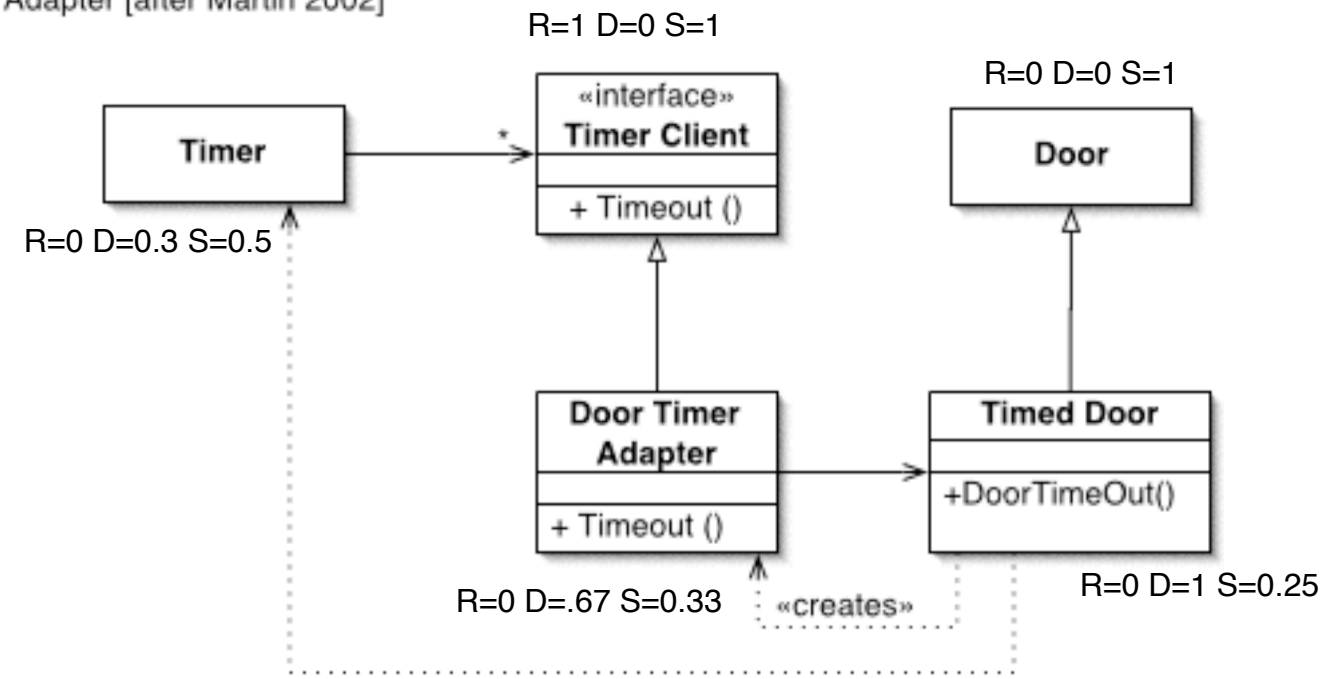


Door Timer Adapter [after Martin 2002]



max A = 2
min A = 1
max E = 3
min E = 0

Door Timer Adapter [after Martin 2002]



max A = 2
min A = 1
max E = 3
min E = 0

Open-Closed Principle

[Bertrand Meyer]

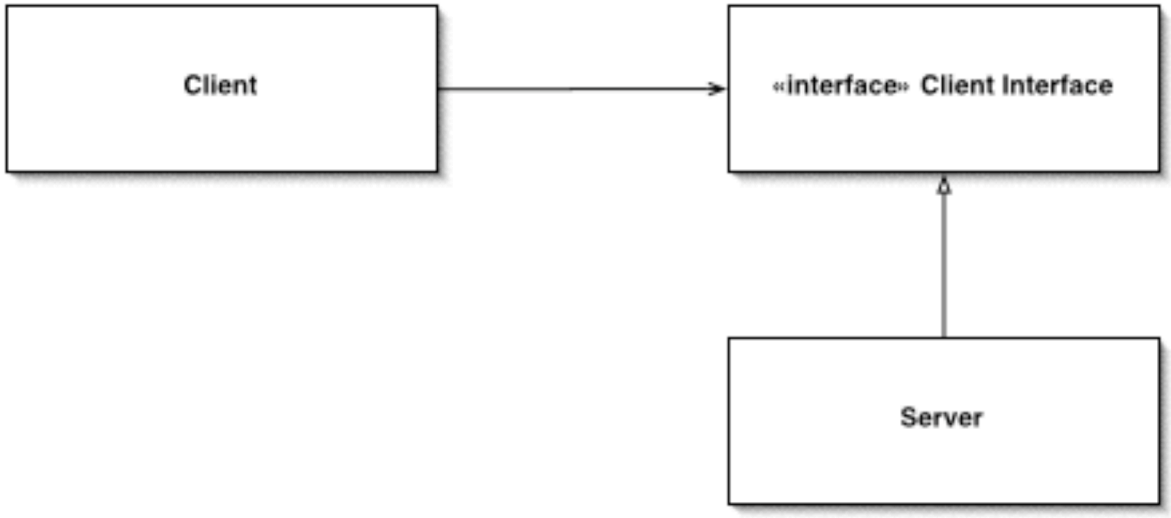
OCP

Software entities should be open for extension but closed for modification.

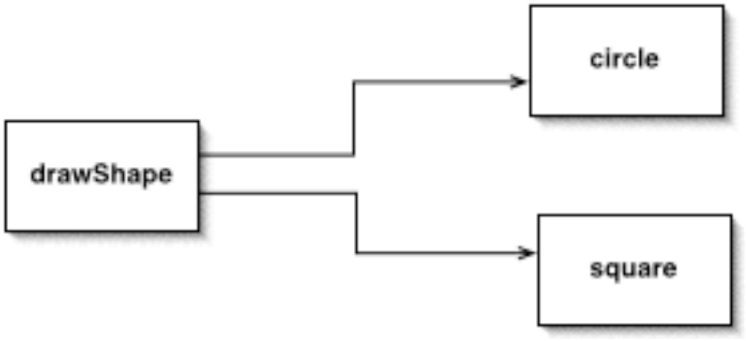
add new behaviours
extend ‘what the entity can do’
extension points

no change to source code
no need to understand internals
(no need to recompile or even relink)
(no need to change source entity)

rigid:
replacing the server
requires change to client



flexible:
replacing the server
requires no change to client

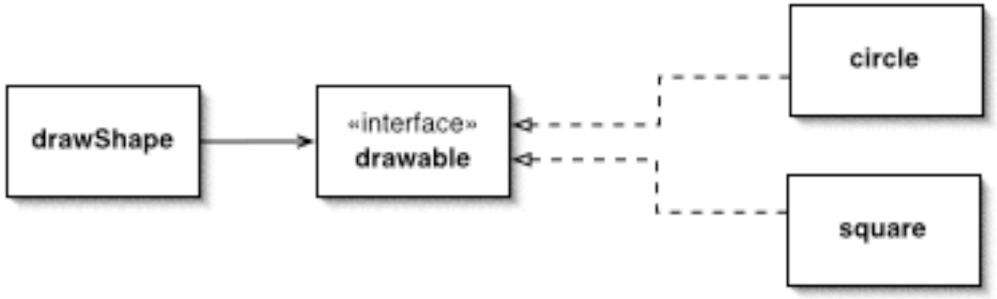


rigid: adding new shapes requires change to draw

fragile: change to draw is spread all over the code

immobile: can't be moved (reused) without reusing shapes

```
-- shape_type.h --------------------------------
enum shape_type {circle, square};

-- circle.h ------------------------------------
struct circle {
     enum shape_type type;
     double radius;
     point centre;
};
void draw_circle (struct circle *);

-- square.h ------------------------------------
struct square {
     enum shape_type type;
     double side;
     point top_left;
};
void draw_square (struct square *);

-- draw_shapes.c -------------------------------
#include "shape_type.h"
#include "circle.h"
#include "square.h"

void drawShapes (void *list [], int n)
{
     int i;
     for (i = 0; i < n; ++i)
     switch (* (enum shape_type *) list [i])
     {
     case square: draw_square ( (struct square *) list [i] ); break;
     case circle: draw_circle ( (struct circle *) list [i] ); break;
     }
}
```


drawShape
«interface»
drawable
circle
square

```
-- drawable.h -------------------------------
/* Add an entry for each new shape type */
DRAWABLE(circle,struct circle,draw_circle)
DRAWABLE(square,struct square,draw_square)

-- shape_type.h -----------------------------
enum shape_type {
#    define DRAWABLE(type_tag,type,draw_function) type_tag,
#    include "drawable.h"
#    undef DRAWABLE
};

# define DRAWABLE(type_tag,type,draw_function) void draw_function(void *);
# include "drawable.h"
# undef DRAWABLE

extern void (*draw_functions []) (void *);

-- shape_type.c -----------------------------
void (*draw_functions []) (void *)= {
#       define DRAWABLE(type_tag,type,draw_function) draw_function,
#       include "drawable.h"
};

-- draw_shapes.c ----------------------------

void drawShapes (void *list [], int n)
{
     int i;
     for (i = 0; i < n; ++i) {
          shape_type t = * (enum shape_type *) list [i];
          draw_function [t] (list [i]);
     }
}
```

```
-- drawable.h --------------------------------
struct Drawable {
     virtual void draw () = 0;
}

-- circle.h ----------------------------------
struct Circle: public Drawable {
     double radius;
     point centre;
     void draw ();
};

-- square.h ----------------------------------
struct Square: public Drawable {
     double side;
     point top_left;
     void draw ();
};


-- draw_shapes.c -----------------------------

void drawShapes (Drawable *list [], int n)
{
     int i;
     for (i = 0; i < n; ++i)
          list [i]-> draw ();
     }
}
```

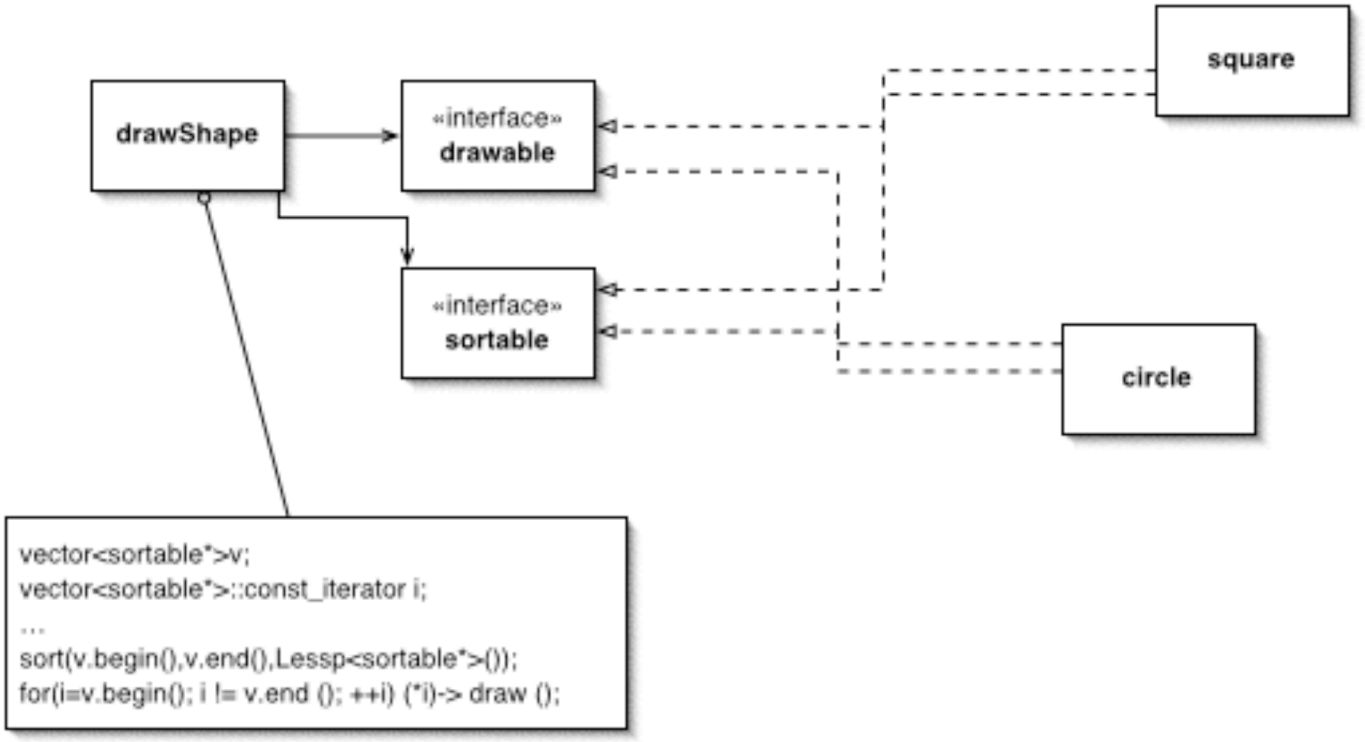
Designing for Change, strategically

We can't anticipate all possible changes.

extension point: anticipated change region

But … **wait** until the extension point is needed.

e.g. “Draw circles first, then squares.”


square
drawShape
«interface»
drawable
«interface»
sortable
circle
vector<sortable*>v;
vector<sortable*>::const_iterator i;
...
sort(v.begin(),v.end(),Lessp<sortable*>());
for(i=v.begin(); i != v.end (); ++i) (*i)-> draw ();

Dependency Inversion Principle

[Robert C Martin]

DIP

Depend upon abstractions

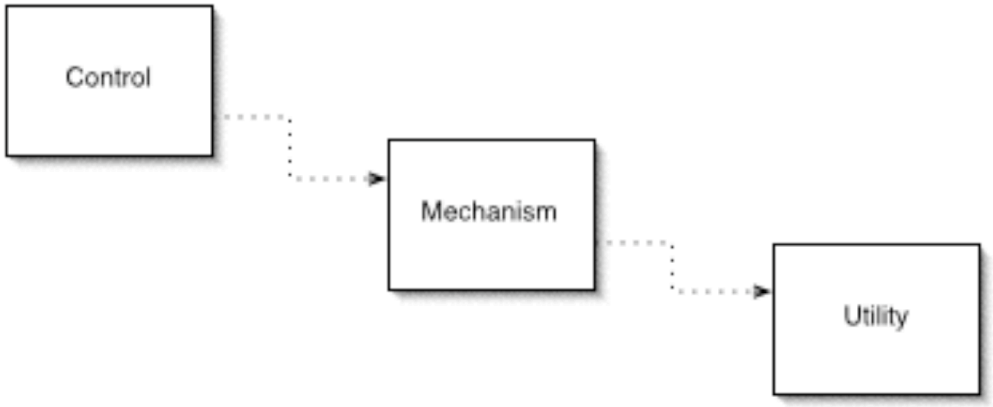
abstractions are more stable

policy and control shouldn't have to change

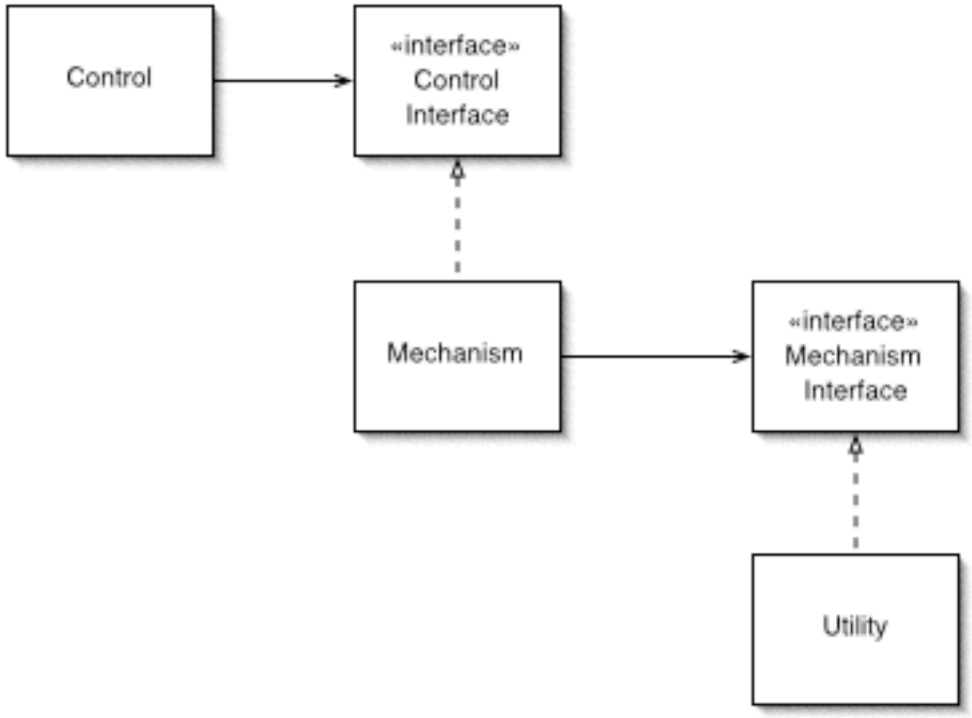
policy and control should be reusable

-> let them define the abstractions they assume and need

Naive layering scheme [Martin, p 128]



Inverted layering scheme [Martin, p 129]



Abstract (high-level, control, policy) is no longer dependent on Concrete (low-level, mechanistic, utility).

Dependency Inversion Heuristics

Depend on Abstractions

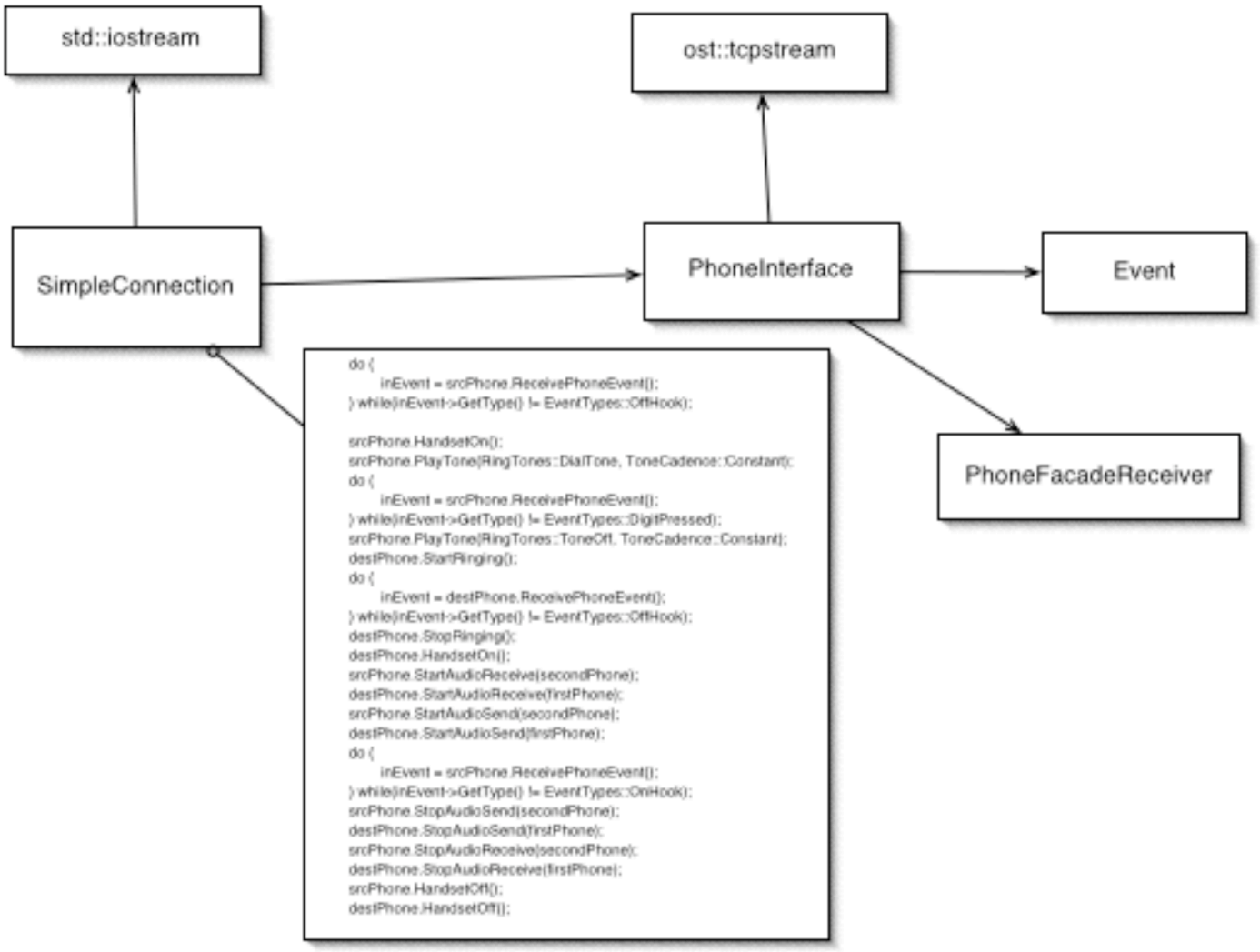
so suspect:

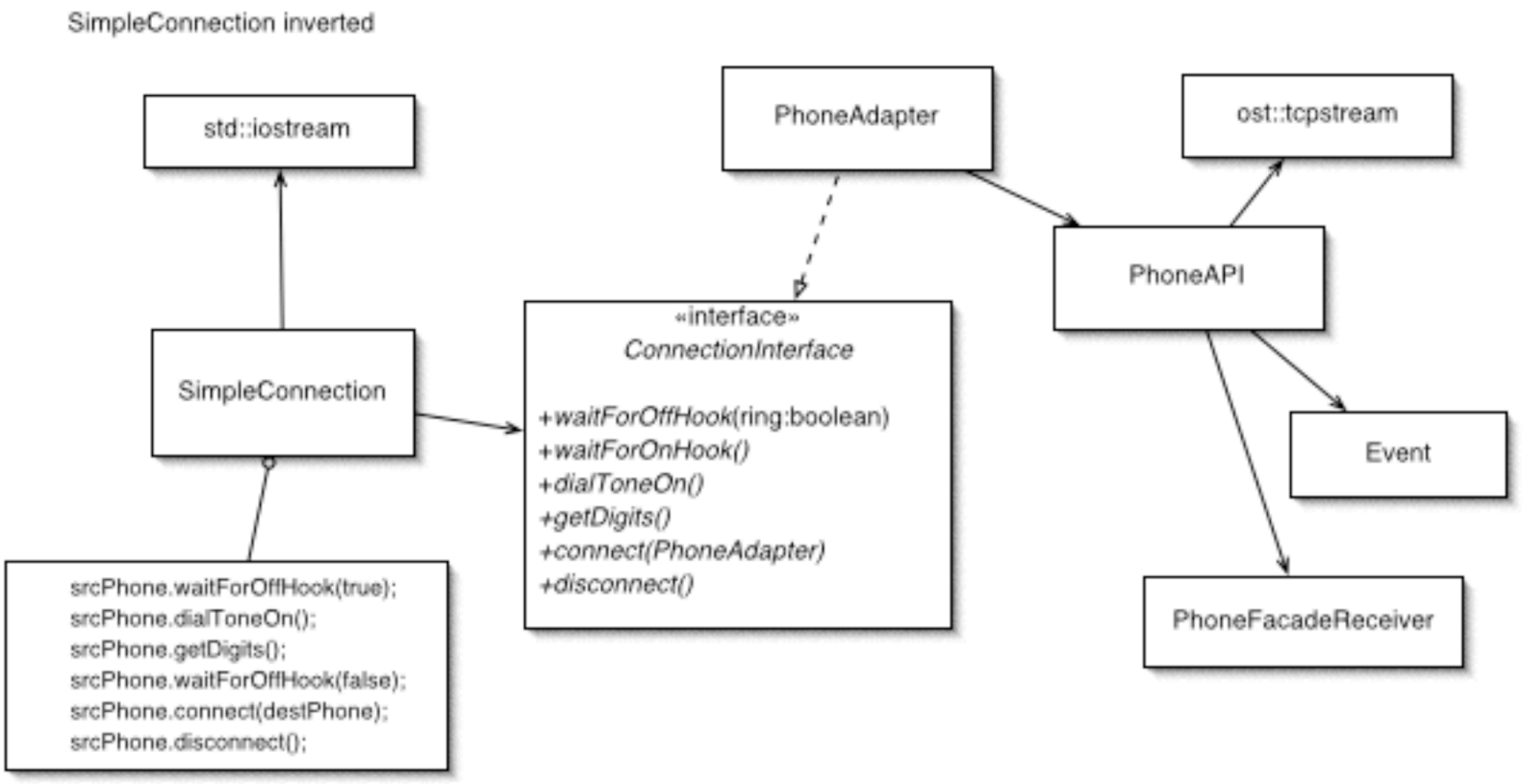
- member variables which refer to concrete classes
- subclasses of concrete classes
- overriding of concrete methods

Depend on stability; so during development, refactoring, and evolution:

- discover the **client's** interface needs
- make independent abstractions, or as part of client
- client logic works through its interface
- server conforms to (implements) interface

SimpleConnection and PhoneInterface [Mennie & Gonsalves 2002]




SimpleConnection inverted
std::iostream
PhoneAdapter
ost::tcpstream
PhoneAPI
SimpleConnection
«interface»
ConnectionInterface
+waitForOffHook(ring:boolean)
+waitForOnHook()
+dialToneOn()
+getDigits()
+connect(PhoneAdapter)
+disconnect()
Event
PhoneFacadeReceiver
srcPhone.waitForOffHook(true);
srcPhone.dialToneOn();
srcPhone.getDigits();
srcPhone.waitForOffHook(false);
srcPhone.connect(destPhone);
srcPhone.disconnect();

Interface Segregation Principle

[Robert C. Martin]

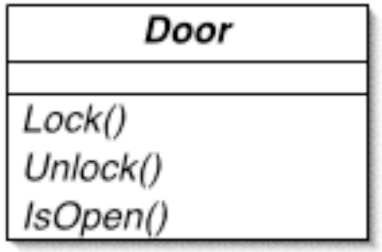
ISP

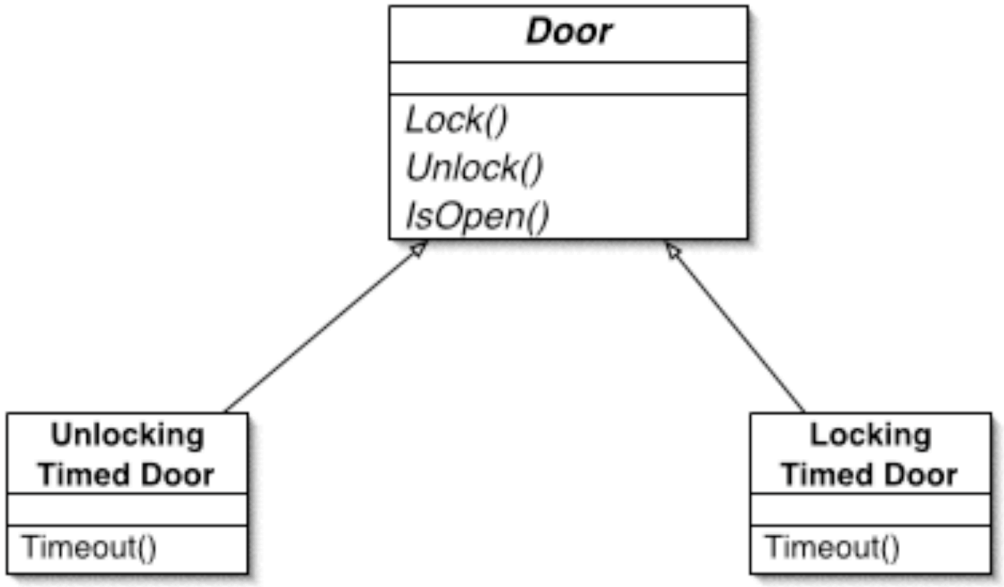
Clients should not be forced to depend on methods which they do not use.

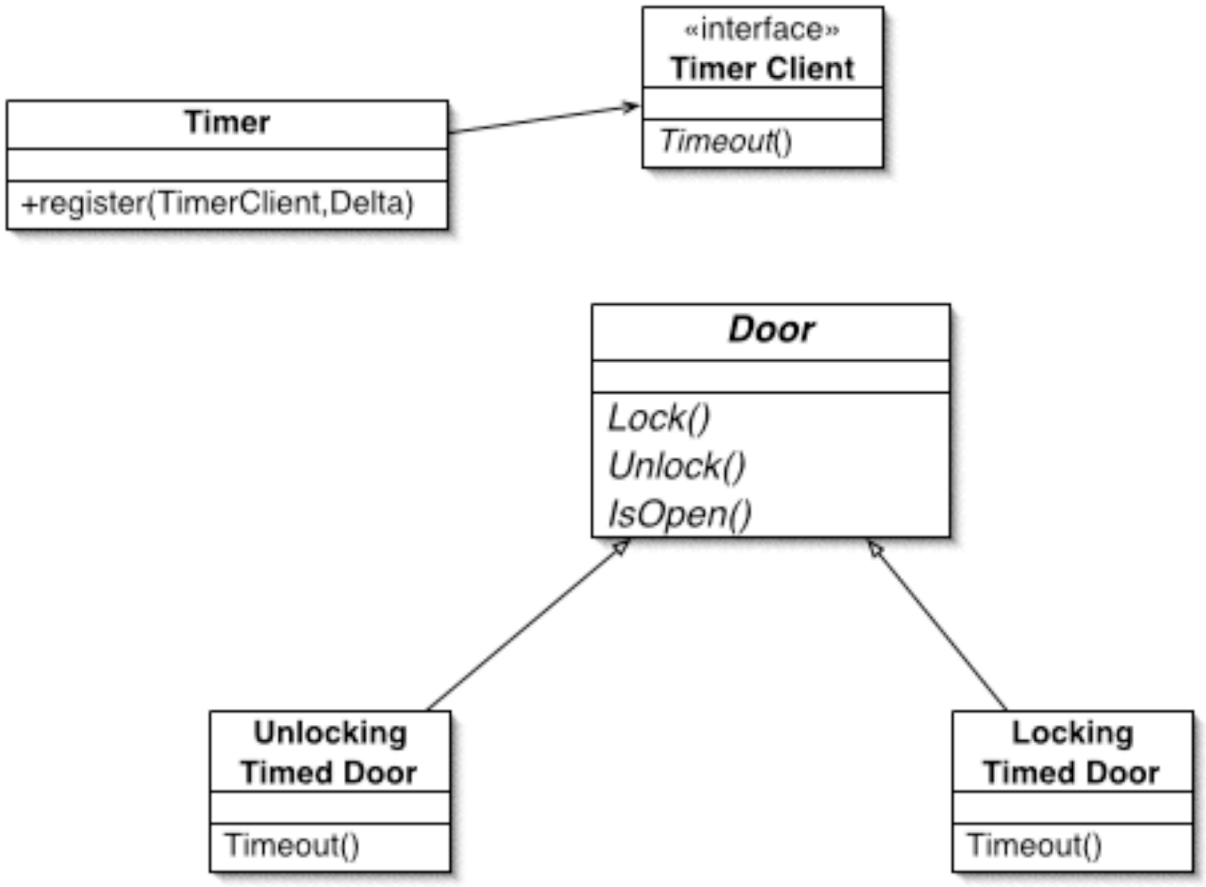
abstractions help to enforce DIP and OCP

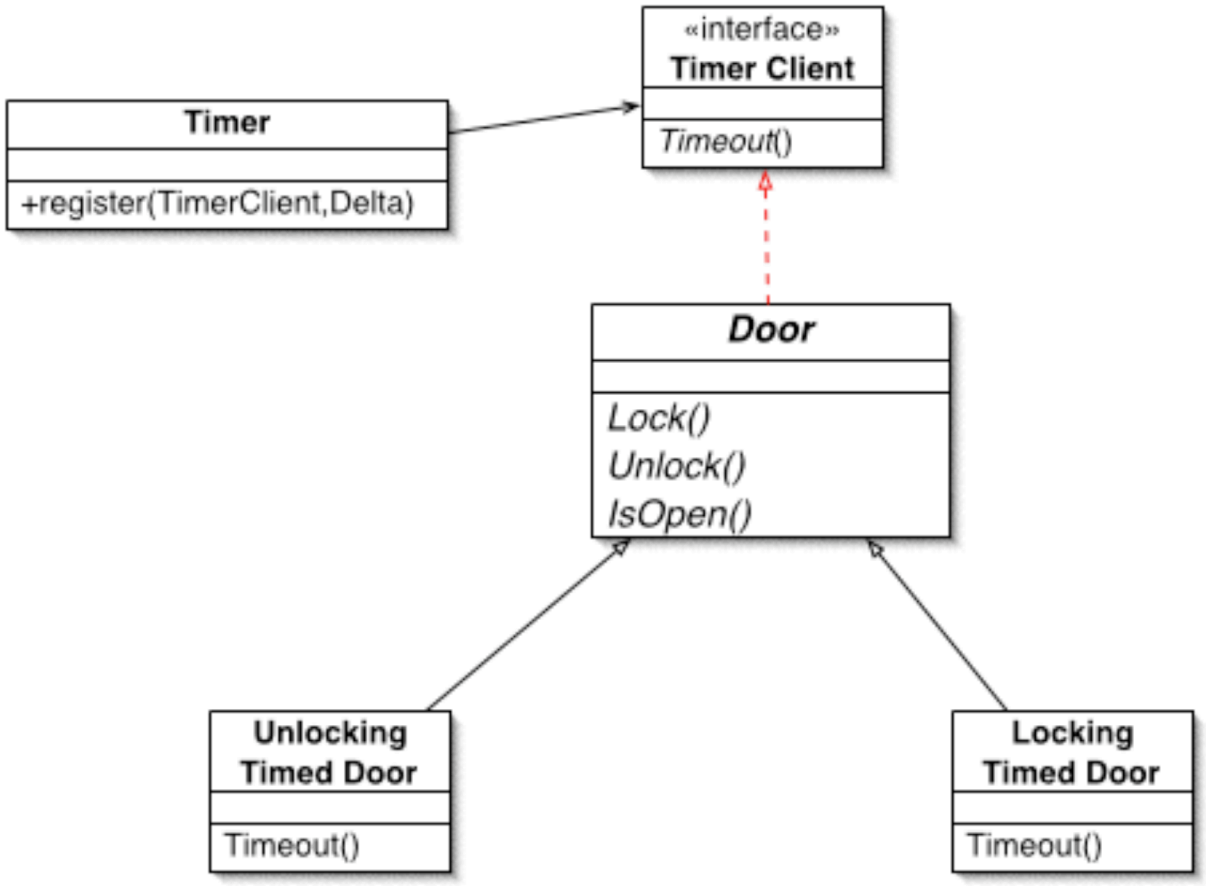
but abstractions have to be cohesive

No “fat” interfaces!

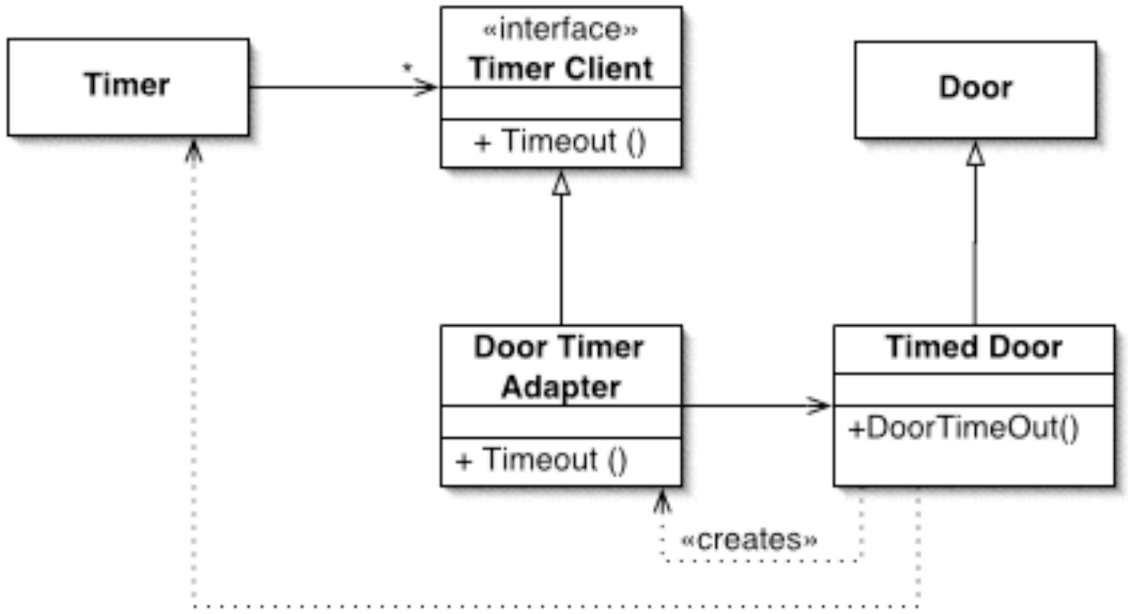

Door
Lock()
Unlock()
IsOpen()


Door
Lock()
Unlock()
IsOpen()
Unlocking Timed Door
Timeout()
Locking Timed Door
Timeout()


«interface»
Timer Client
Timeout()
Timer
+register(TimerClient,Delta)
Door
Lock()
Unlock()
IsOpen()
Unlocking
Timed Door
Timeout()
Locking
Timed Door
Timeout()


«interface»
Timer Client
Timeout()
Timer
+register(TimerClient,Delta)
Door
Lock()
Unlock()
IsOpen()
Unlocking
Timed Door
Timeout()
Locking
Timed Door
Timeout()

Door Timer Adapter [after Martin 2002]

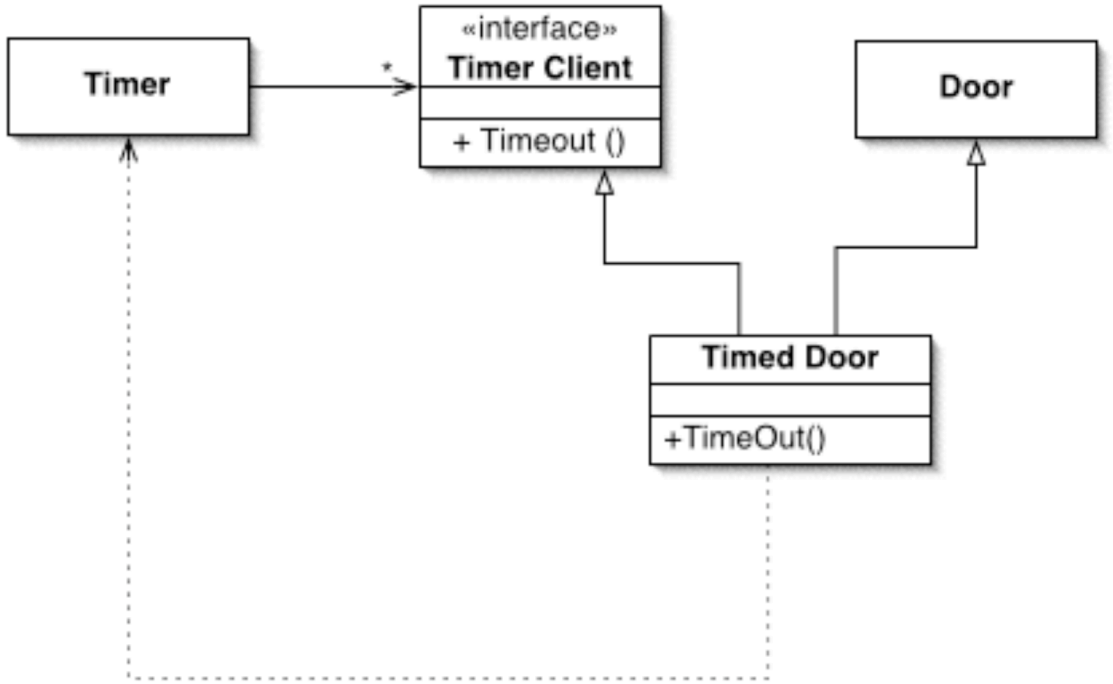


Dynamic solution.

Pattern: adapter

Single inheritance; overhead.

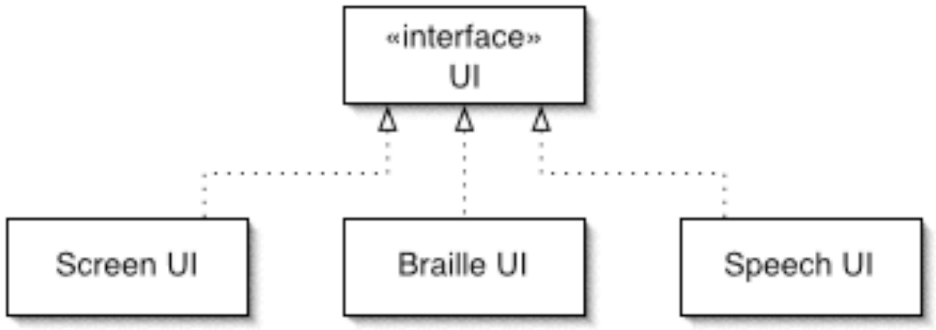
Door Timer Multiple Inheritance [after Martin 2002]



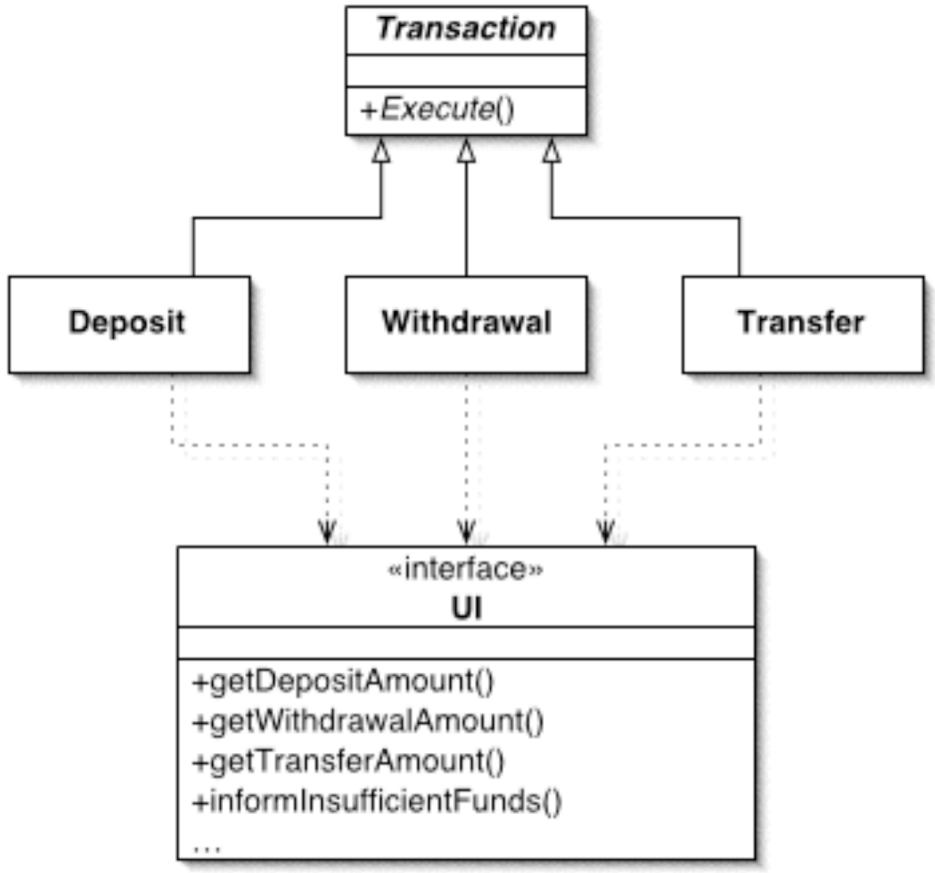
Static solution.

Multiple inheritance.

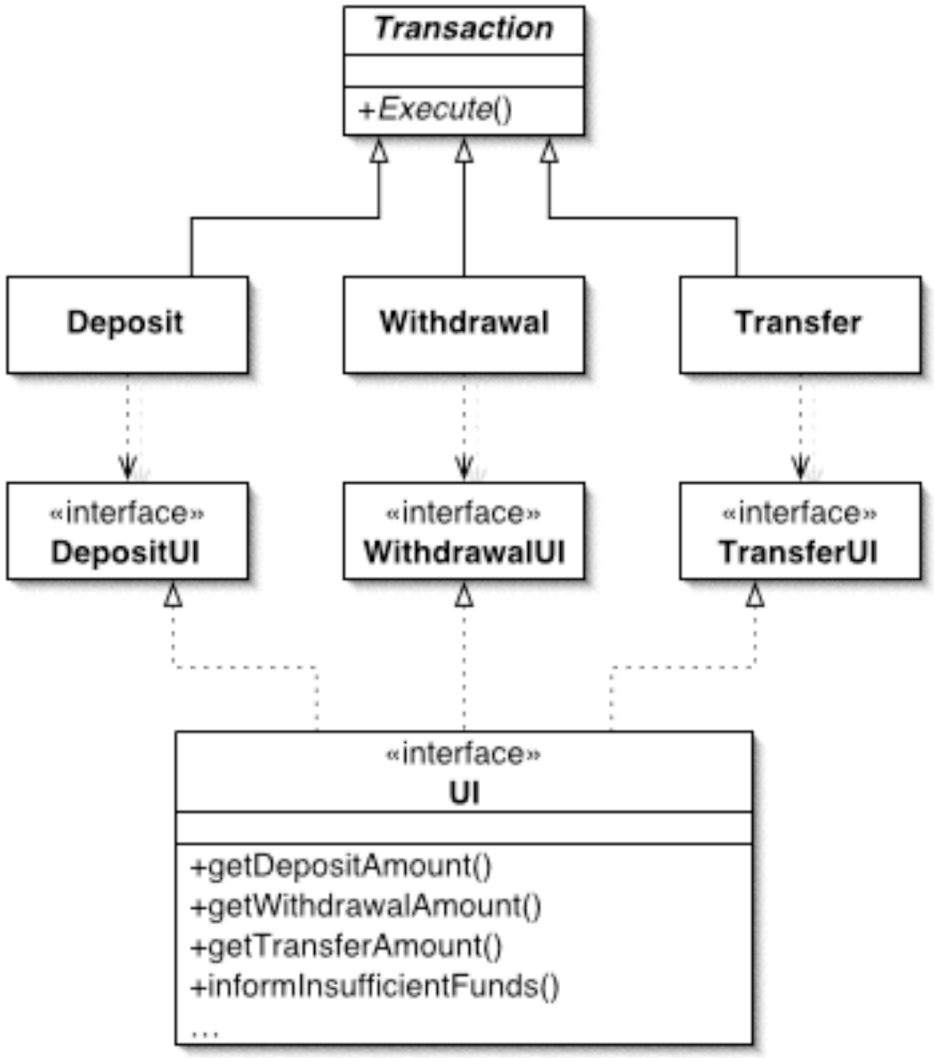
ATM User Interfaces [after Martin]



ATM Transactions [Martin p 140]



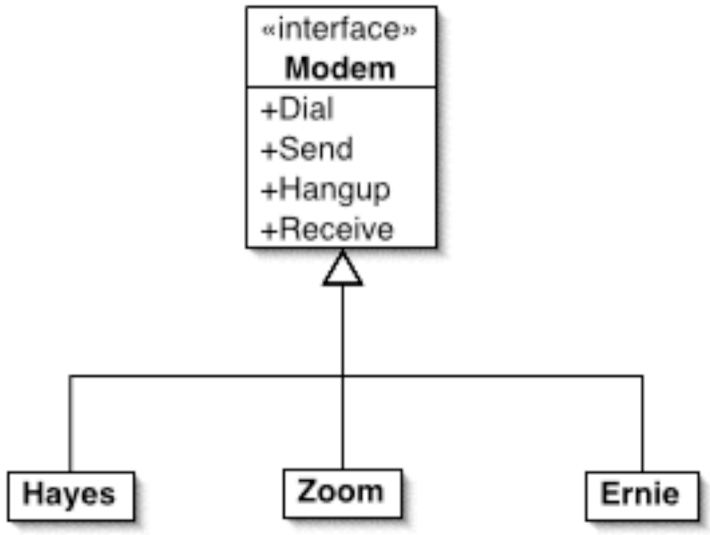
ATM Segregated [Martin p 140]



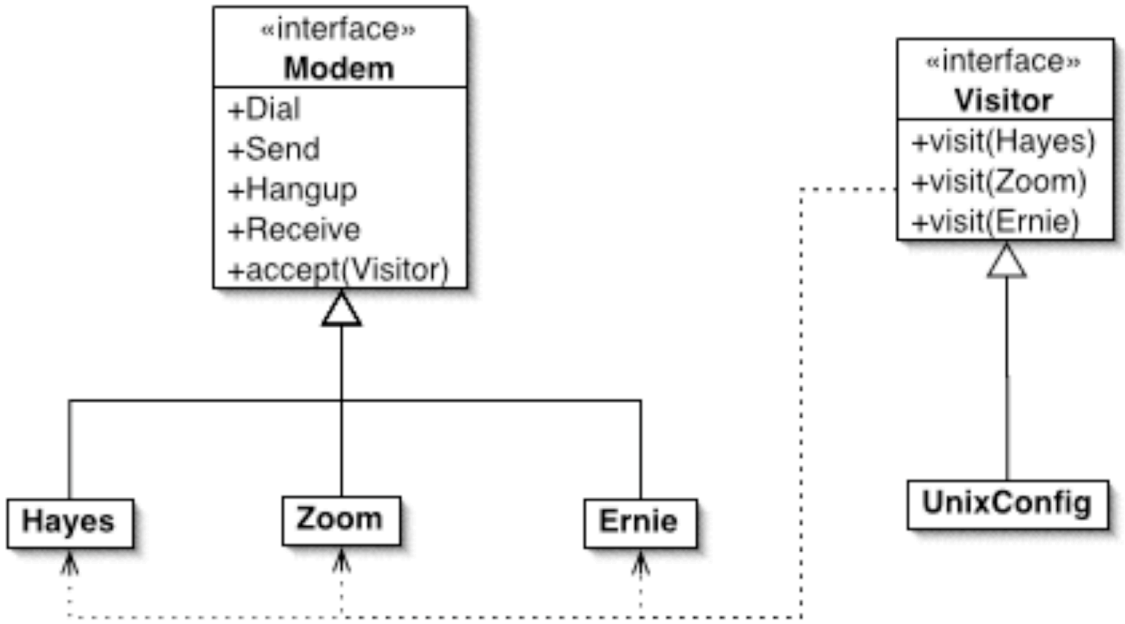
```
-- ui.cc -------------
#include <depositUI.h>
#include <withdrawalUI.h>
#include <transferUI.h>
namespace UIGlobals      // can't be class, must be namespace!
{
     static UI theUI;
     DepositUI & depositUI = theUI;
     WithDrawalUI & withdrawalUI = theUI;
     TransferUI & transferUI = theUI;
}
…

-- deposit.{h,cc} -------------

#include <depositUI.h>
namespace UIGlobals
{
     extern DepositUI depositUI;
}

Deposit::Execute()
{
     …
     DepositUI & ui = UIGlobals::depositUI;
     …
     ui. getDepositAmount ();
     …
}
```

Closing an inheritance hierarchy
(left open for extension)

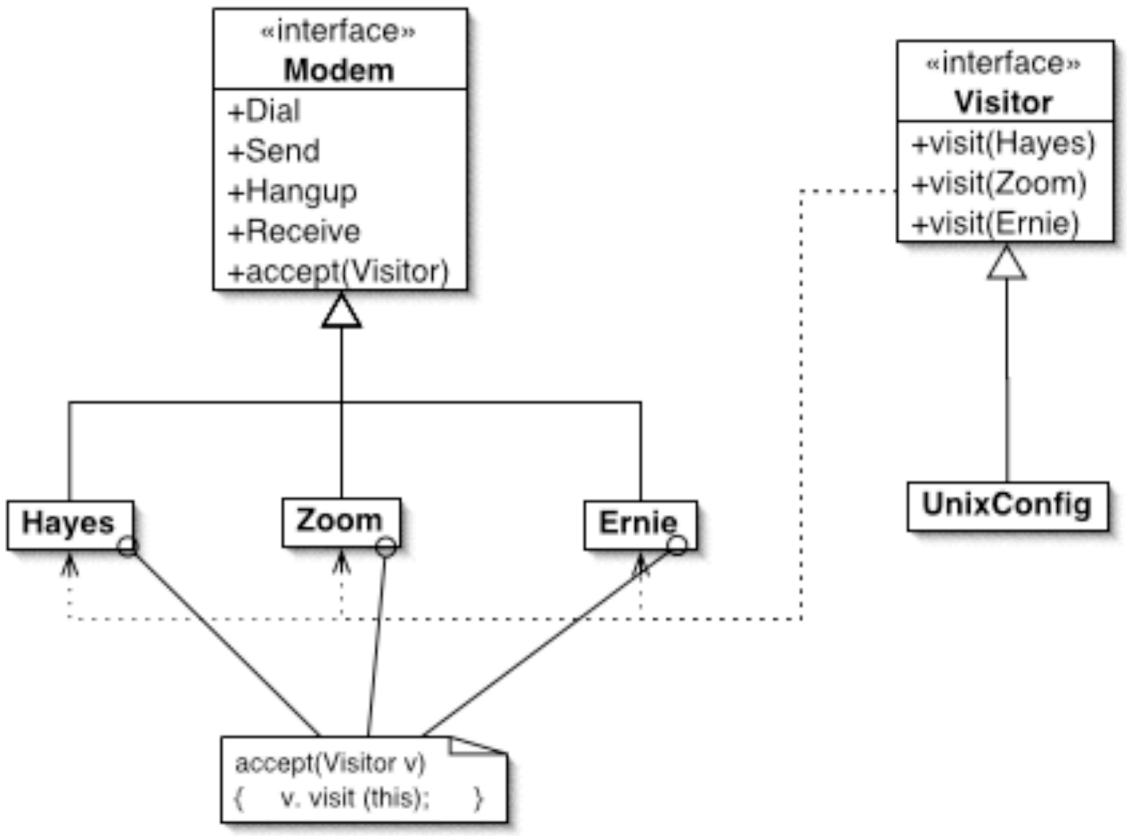
Modem Hierarchy [Martin]

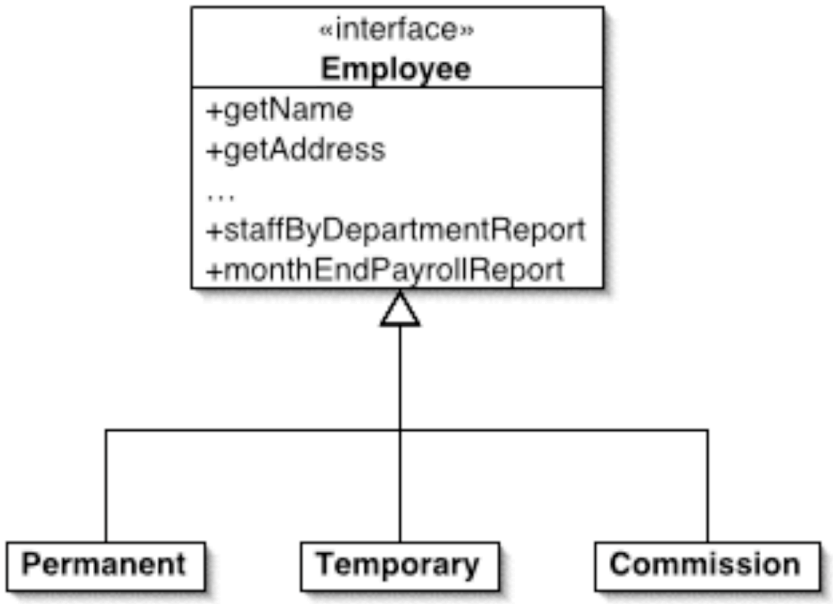


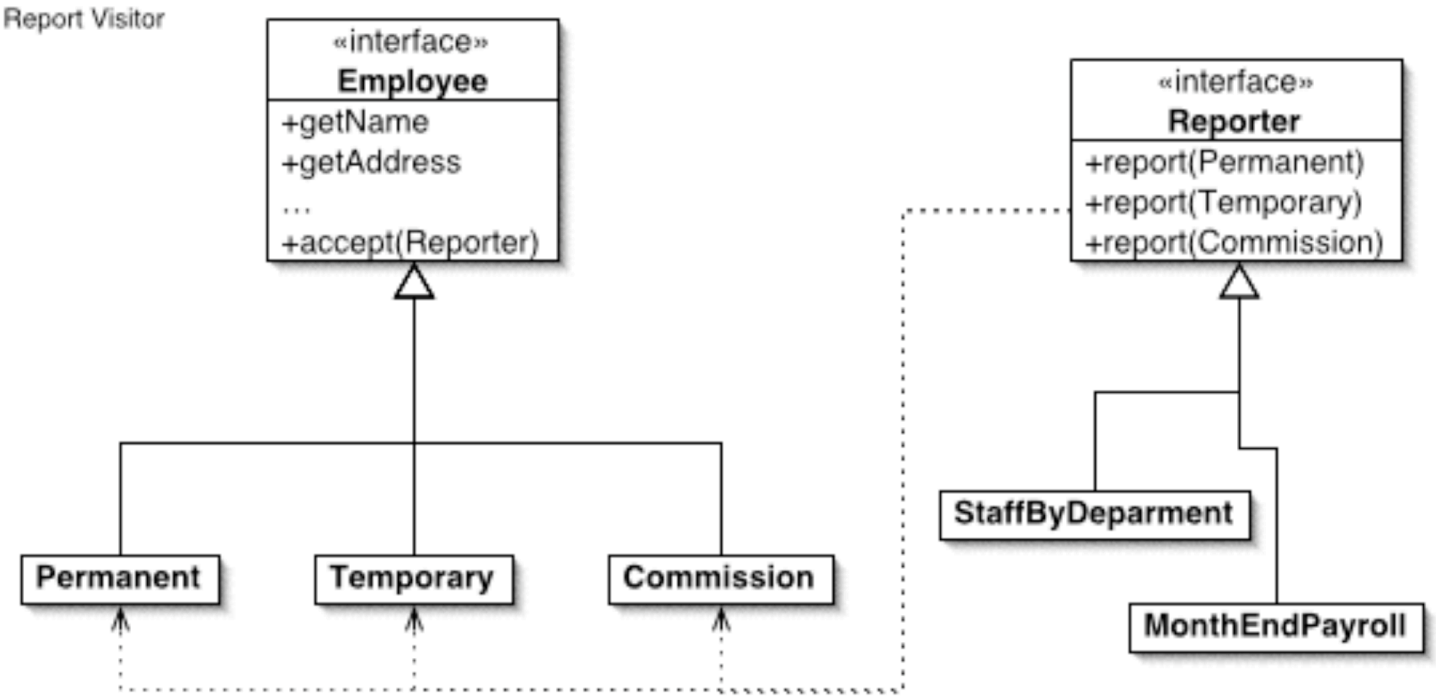
Modem Visitor [Martin]



Modem Visitor [Martin]

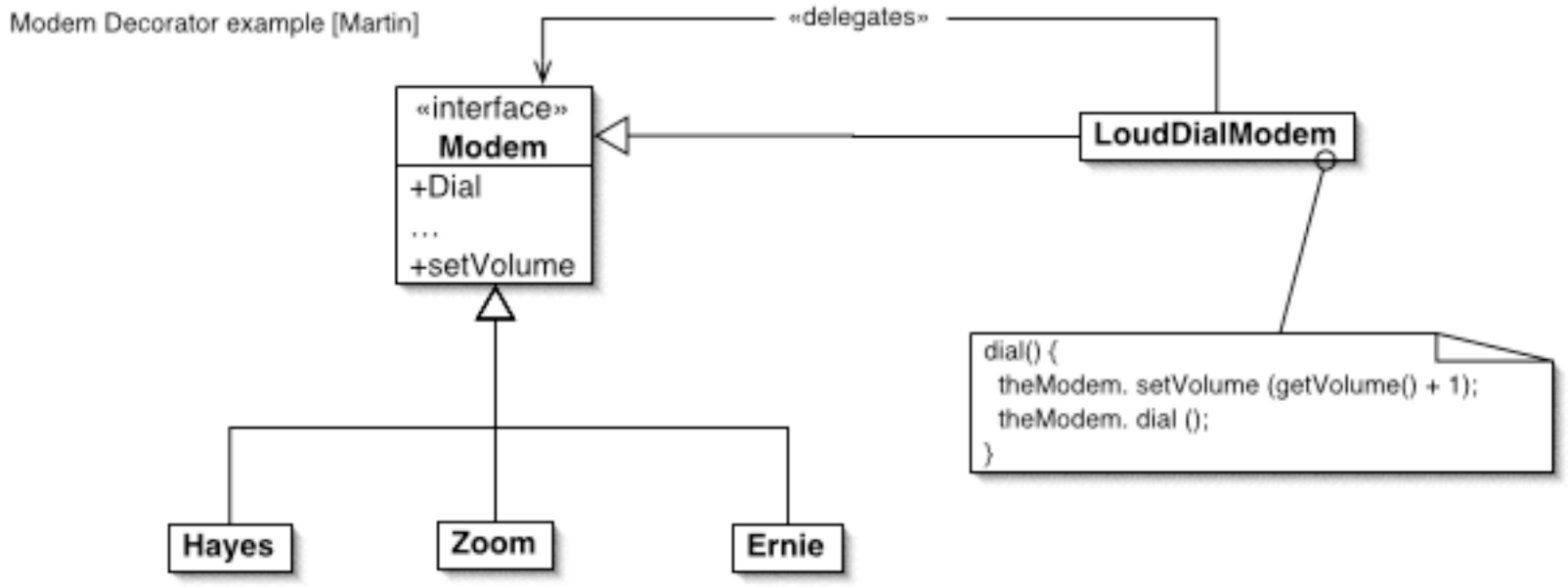



«interface»
Employee
+getName
+getAddress
...
+staffByDepartmentReport
+monthEndPayrollReport
Permanent
Temporary
Commission


Report Visitor
«interface»
Employee
+getName
+getAddress
…
+accept(Reporter)
«interface»
Reporter
+report(Permanent)
+report(Temporary)
+report(Commission)
Permanent
Temporary
Commission
StaffByDeparment
MonthEndPayroll

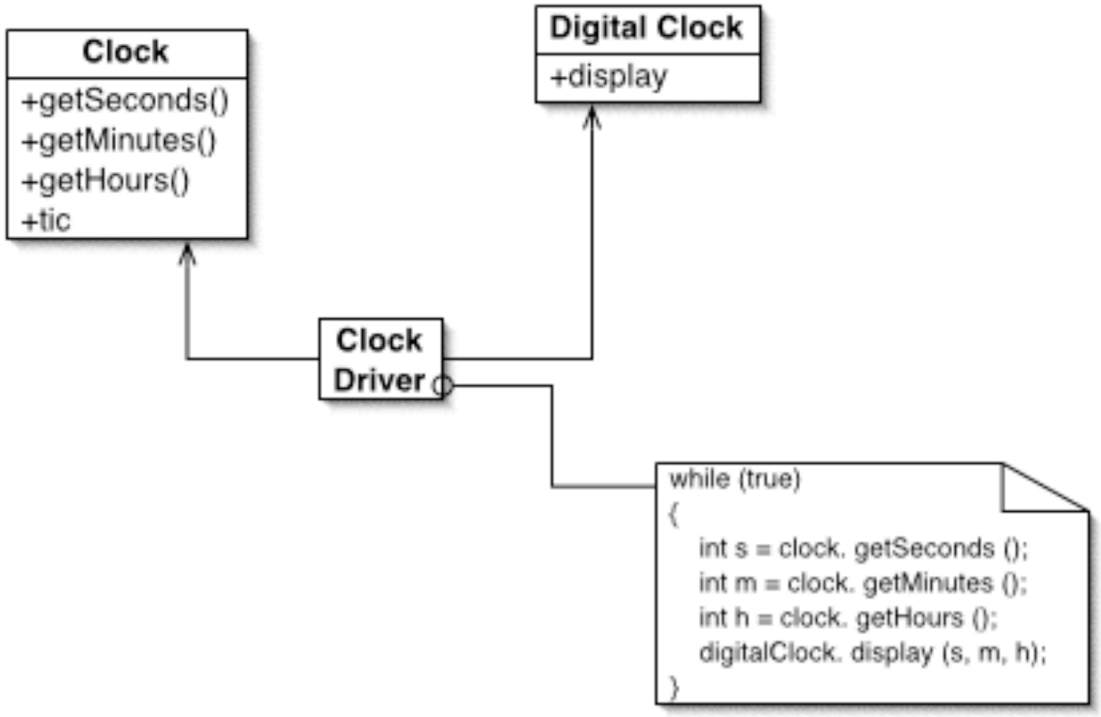
Closing an inheritance hierarchy
(left open for extension)

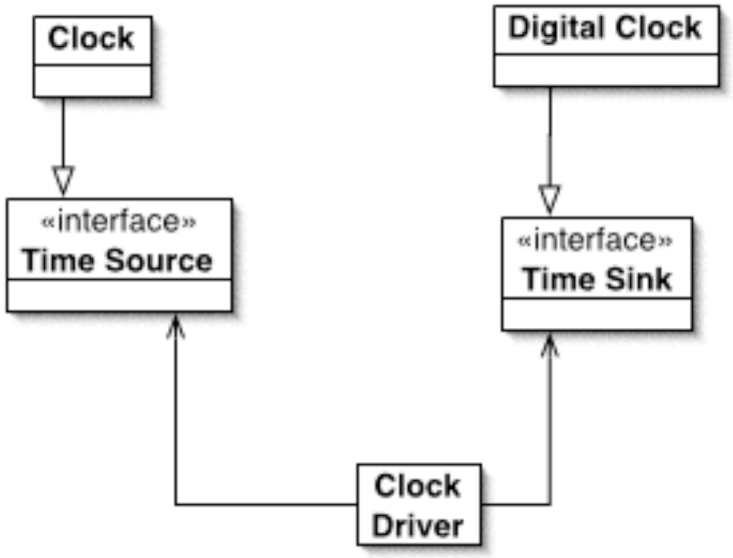
- **Decorator**



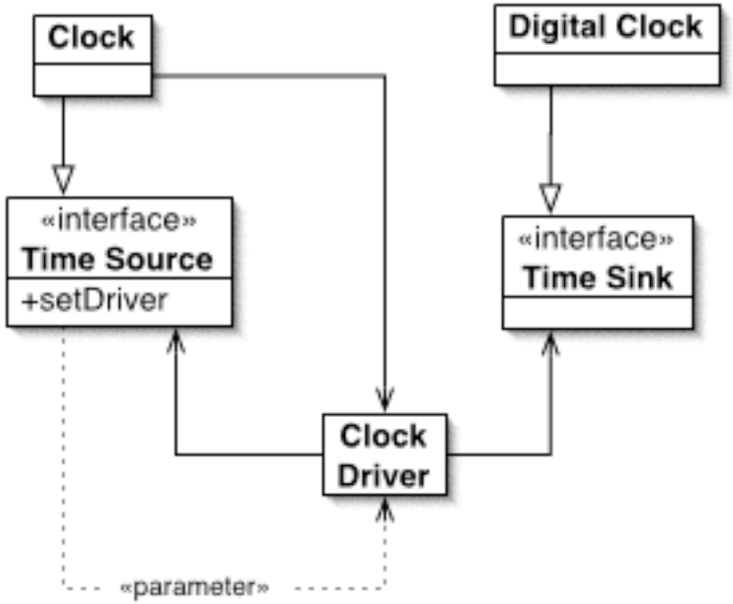
- **Multiple Decorator**
- **Extension Object**
- **Proxy**

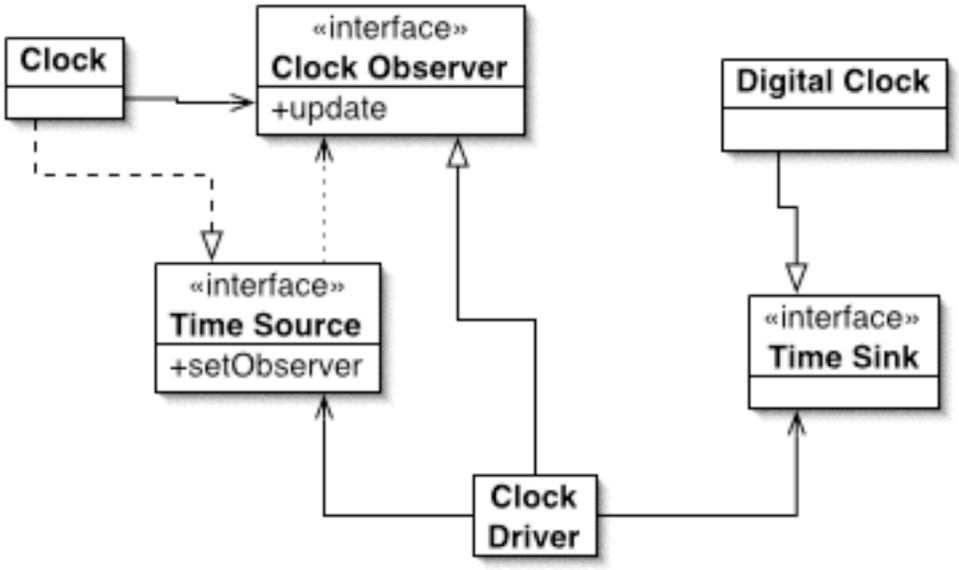
Closing event handling
(left open for event type)



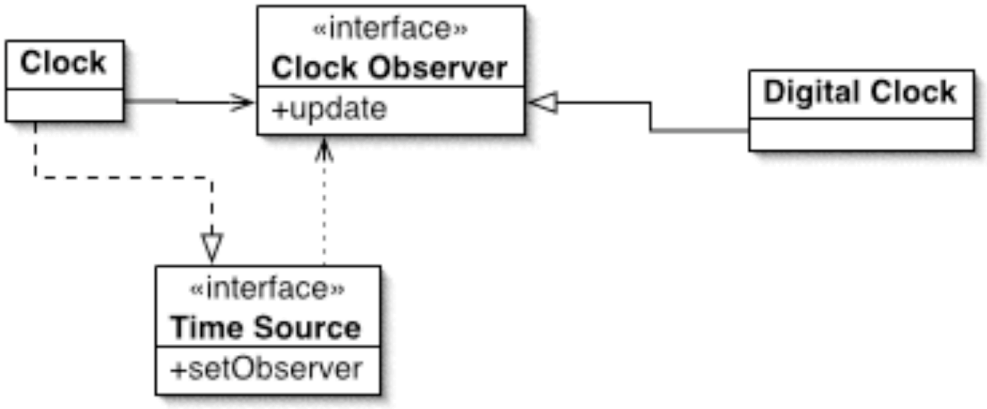

Clock
Digital Clock
«interface»
Time Source
«interface»
Time Sink
Clock
Driver

Not abstract enough:
- Time Source still depends on Driver as parameter
- Clock uses Clock Driver (internally) after setting.

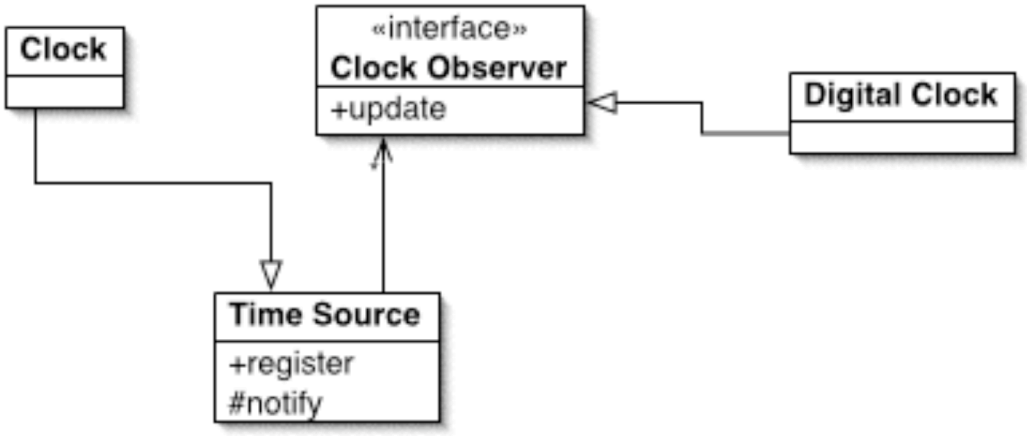




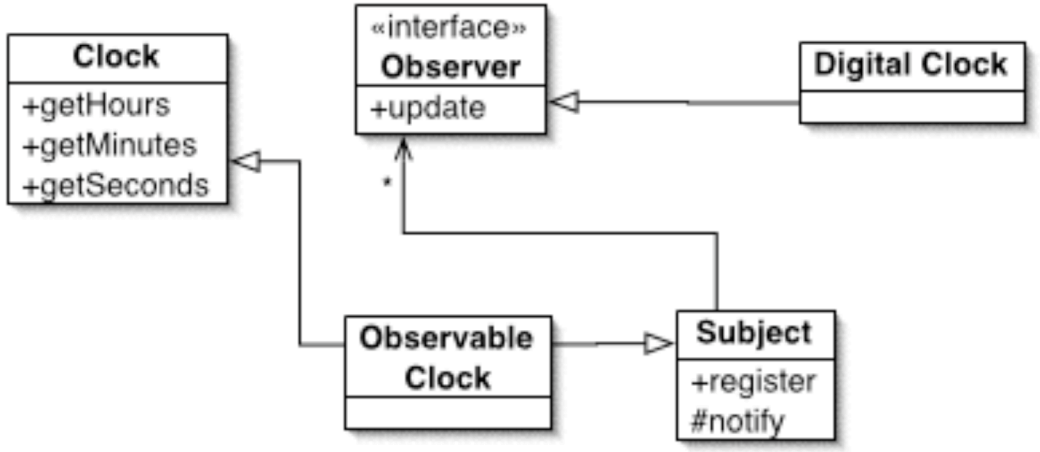
- Now Time Source is reusable
- Clock no longer uses (concrete) Clock Driver



- Didn't really need the Clock Driver...



- Adding multiple Observers
- But Clock has to contain code to handle registration and dispatch (not cohesive)



- solution using multiple inheritance
- exercise: do this using Java