Menagerie of Dispensers

One button operation

A `press` operation returns a new ticket number:

```
type Dispenser = interface {
  press -> Number
}

class dispenser -> Dispenser {
  var count : Number := 0
  method press -> Number {
    count := count + 2
    count
  }
}
```

Result must be even

Hoare logic style:

```
d : Dispenser { r = d.press } even(r)
```

Chainmail style:

```
forall d : Dispenser. forall o : Object
  [ o.calls {r = d.press} --> Next(even(r)) ]
```

Result must be monotonically increasing

Chainmail v1:

```
forall d : Dispenser [ even(d.count) && [ Next(d.count == c') --> (c' >= d.count)
] ]
  // requires d.count as ghost field

forall c : Number, d : Dispenser [ d.count == c { r = d.press } r == c + 2 & d.cou
nt = r ]
  //Hoare tripple in the middle?
```

Chainmail v2:

```
forall n, n' : Number, forall d : Dispencer [ n==(d.press) --> (even(n) && [Next(n
'==d.press) --> n' >= n])]
```

Revocable

```
type RevocableDispenser = interface {
press -> Number
revoke
}

class revocableDispenser {
  var count : Number := 0 is ghost //hmm
  var state : Boolean := true is ghost //hmm
  method press {
    if (state) then {
     count := count + 2
     count else {
     error "revoked"
  }
  method revoke {state := false}
  method switch {state := !state}
}
```

Hoare logic version:

```
(d.state = true) && (d.counter = c)  {d.press}  (d.state = true) && (d.counter = c
 + 2)
```

Or, stealing syntax from somewhere I've forgotten:

```
pre  (d.state = true) && (d.counter = c)
prog {d.press}
post (d.state = true) && (d.counter = c + 2)
```

Chainmail version:

```
forall d : Dispenser, o : Object (d.state == true) ! Past(o calls d.revoke)
```

(either way, need to adapt spec of next to deal with errors one way or another)

switchable

```
forall d : Dispenser, s : Boolean.  d.state = s && Next(d.state == ! s) --> exists
 o : Object. [o.calls d.switch]
forall d : Dispenser, s : Boolean.  d.state = s && Will(d.state == ! s) --> Will(e
xists o : Object. [o.calls d.switch]  )
```

The version below uses two extra random bits of notation: * `e @ t` - expression at time t * `Tony(t)` - assuming t is a call, the matching return is just done.

```
(d.state == s) @ t && (d.state ==_t !s) @ Tony(t) --> exists t'' . t < t'' < Tony(
t). exists o [o.calls d.switch] @ t''
```

Two button operation

- A press opration presses the button
- A take operation retrieves the ticket

```
type Dispenser = interface {
press
take -> Number
}

class dispenser -> Dispenser {
  var count : Number := 0
  var pressed : Boolean := false
  method press { pressed := true  }
  method take {
    if (pressed) then { count := count + 2
                        pressed := false
                        count }
                  else { error "nothing will come of nothing" }
  }
}
```

push button, non accumulating

```
d.pressed {r = take} r : Ticket && d.pressed = false
!d.pressed { r = take} r = error
d.pressed && Next(~d.pressed) --> exists o. <o calls d.take>
```

push button, non accumulating

assume additional `var presses := 0 is ghost`

```
forall d' : Number;  d' == d.presses && d' > 0 {r = take} r : Ticket && d.presses
== d' - 1
forall d' : Number;  d' == d.presses && {press} d.presses == d' + 1
d.presses <= 0 { r = take} r = error
forall d' : Number; d.presses && Next(d.presses != d) --> exists o. (<o calls d.ta
ke> || <o calls d.press>)
```

small matters of specifying

- coloured tickets
- price
- delay / timeout
- pin number