Defensive Programming Continued From Last Time…

Possible Error Handling Techniques

- Return a neutral value
- Substitute the next piece of valid data
- Return the same answer as the previous time
- Substitute the closest legal value
- Log a warning message to a file
- Return an error code
- Call an error-processing routine/object
- Display an error message whenever the error is encountered
- Handle the error in whatever way works best locally
- Shut down

Intro to Exceptions

- Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.
- To catch exceptions, a portion of code is placed under exception inspection.
- When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

http://www.cplusplus.com/doc/tutorial/exceptions/

Simple Exception Example

```
// exceptions
#include <iostream>
using namespace std;

int main () {
  try
  {
    // some complex code
    throw 20; // “throw” an exception if the complex code failed
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << endl;
  }
  return 0;
}
```

http://www.cplusplus.com/doc/tutorial/exceptions/

You can “throw” anything: int, string, etc..

```
try {
  throw string("Example Error");
}
catch (string s)
{
  cout << "Error occurred: " << s << endl;
}
```

- Exceptions are a very powerful error handling mechanism.
- They can also drive you crazy… be very careful with the order of execution for exceptions…

“catch” only catches exceptions of the correct type.

```
try {
  line 1 of C++ code
  line 2
  if (line_2_failed) throw 10;
  line 4
  if (line_4_failed) throw ‘f’;
  line 6
  line 7
  if (line_7_failed) throw string(“line 7 failed”);
  line 9
  line 10
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

http://www.cplusplus.com/doc/tutorial/exceptions/

You can have nested try-catch blocks

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Forwards the exception on to the outer block

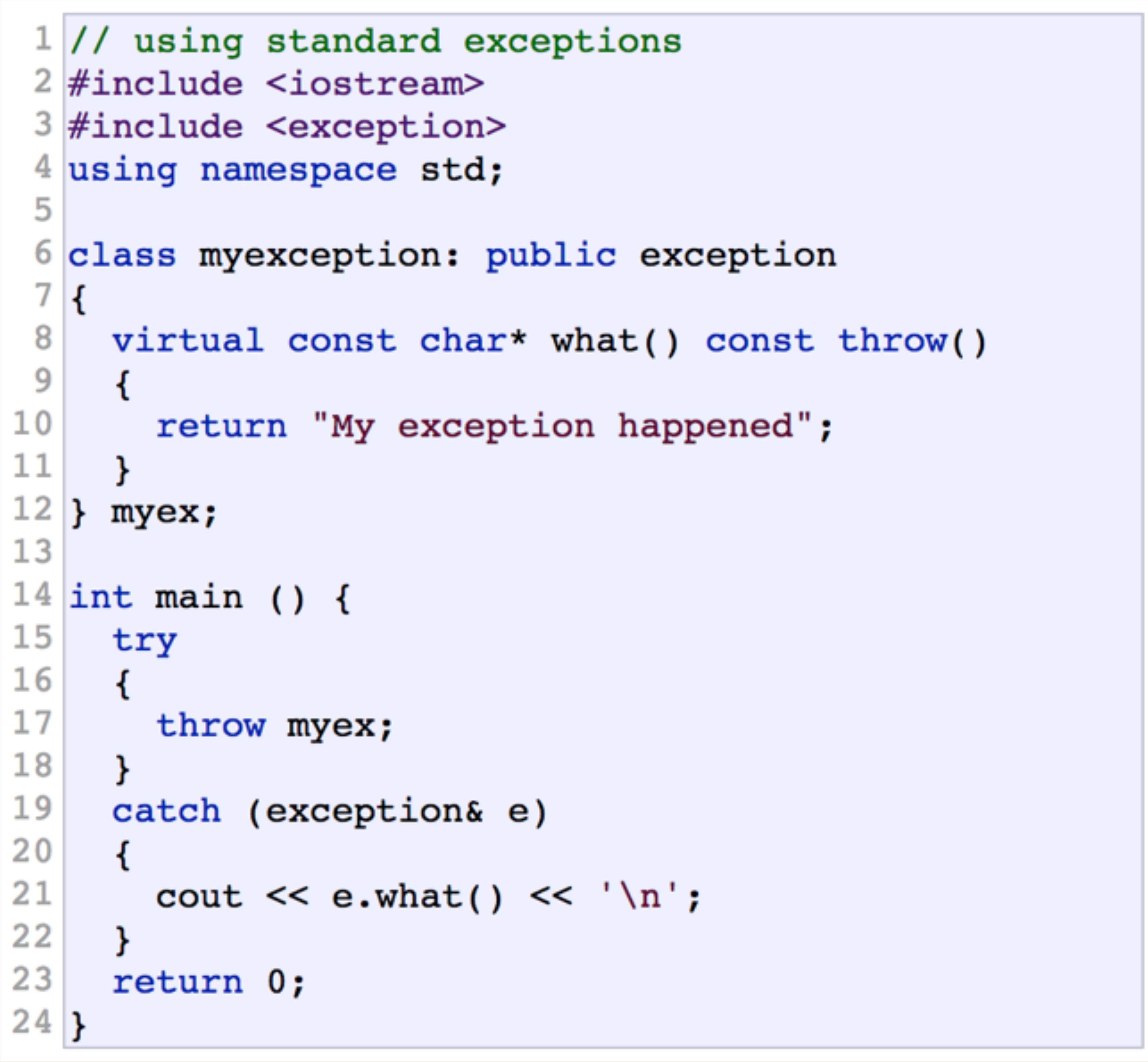
- How could you use this?
- Note how powerful this is… but also possibly confusing. The code no longer runs linearly.

http://www.cplusplus.com/doc/tutorial/exceptions/

The C++ standard library (std)

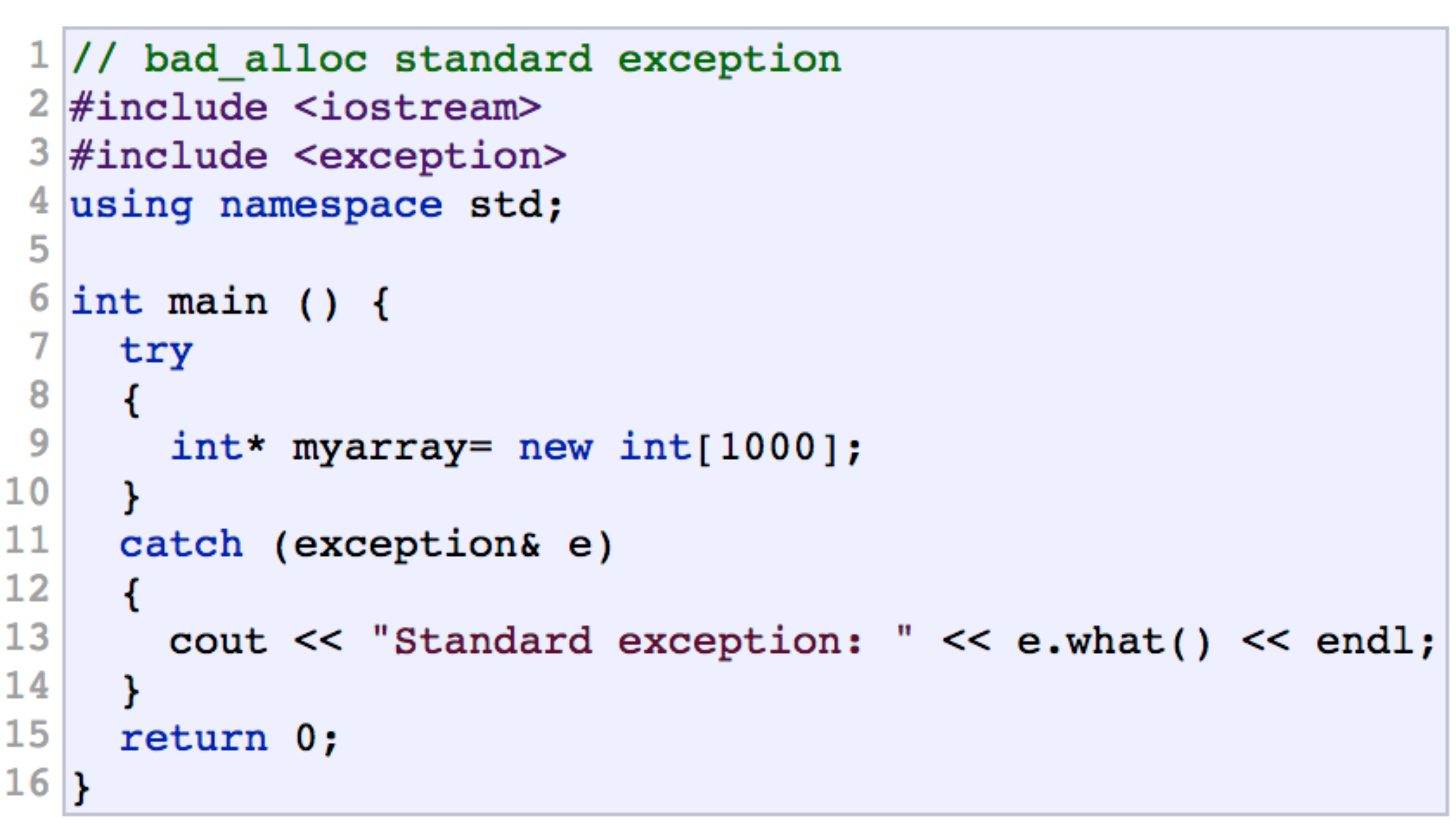
- We’ve used std for several things already:
  - string
  - cout, cerr
- std also includes a useful base class for defining exceptions.
- std uses this class itself for all of its internal error handling.
- you can also extend it and use it yourself.

Here’s how to extend std::exception to create your own “myexception" class.



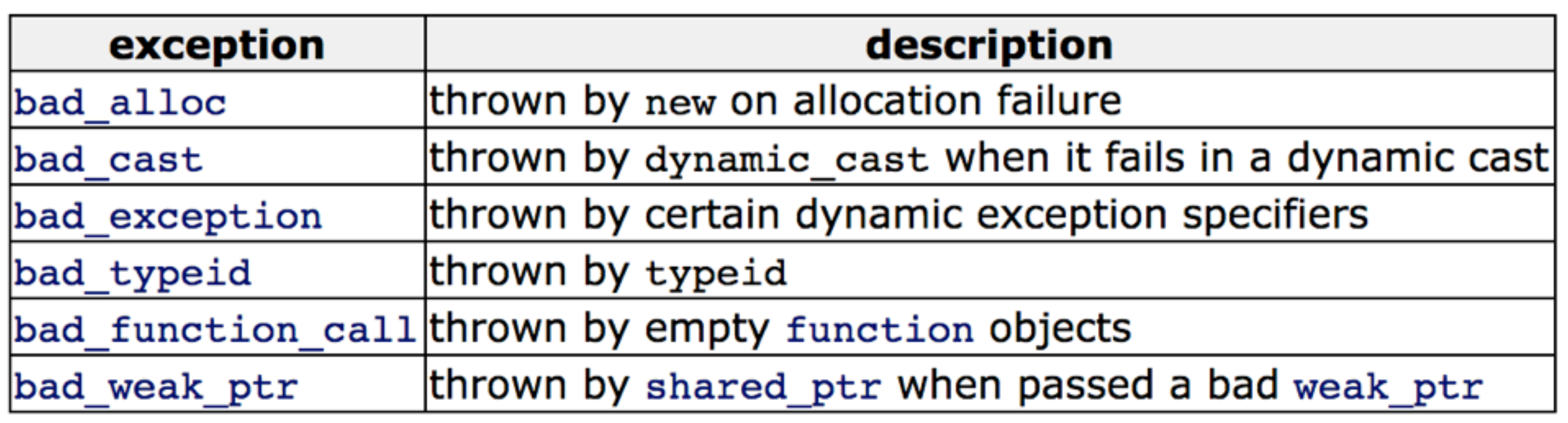
```cpp
// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

Here’s how to catch std::exception’s thrown within the std library:



```cpp
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

Here are the different exceptions that std can throw:



exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_exception	thrown by certain dynamic exception specifiers
bad_typeid	thrown by typeid
bad_function_call	thrown by empty function objects
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr

Remember, exceptions are just a programmatic way to catch errors, you still need to handle them in some way.

- Return a neutral value
- Substitute the next piece of valid data
- Return the same answer as the previous time
- Substitute the closest legal value
- Log a warning message to a file
- Return an error code
- Call an error-processing routine/object
- Display an error message whenever the error is encountered
- Handle the error in whatever way works best locally
- Shut down

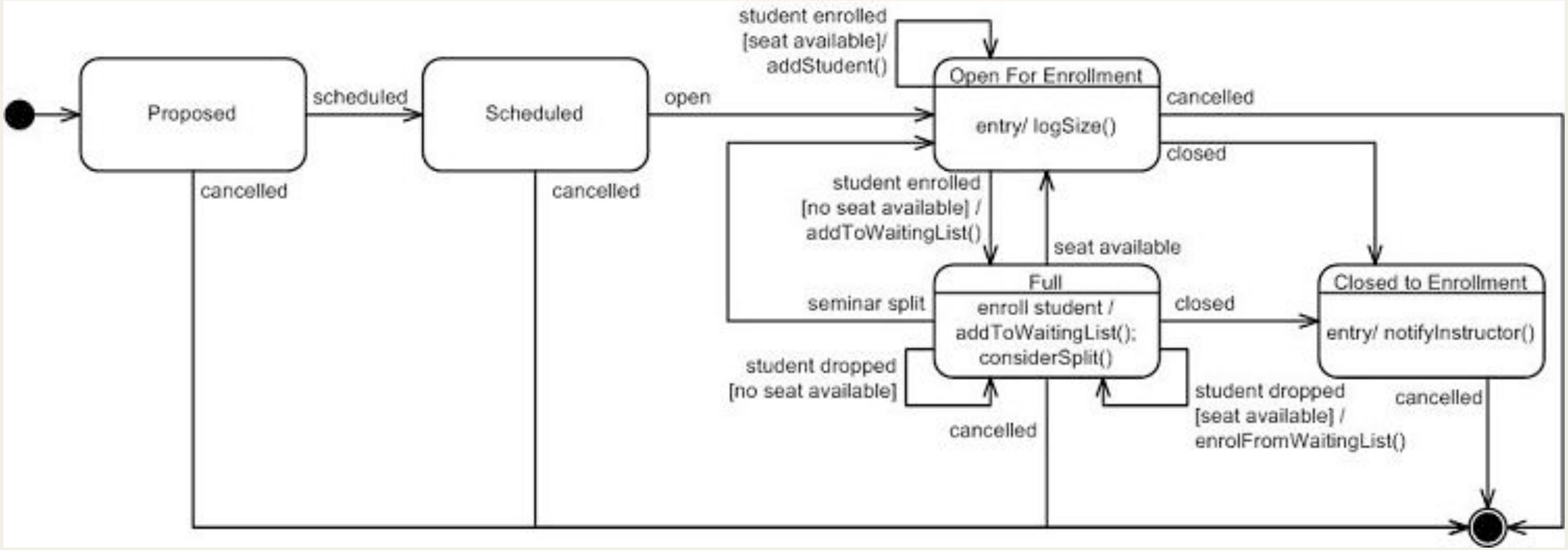
One More Favorite Design Pattern:

State Machines / The State Design Pattern

CSci-3081W: Program Design and Development

Sources for this lecture: Fowler's text and Head First Design Patterns.

State Machines



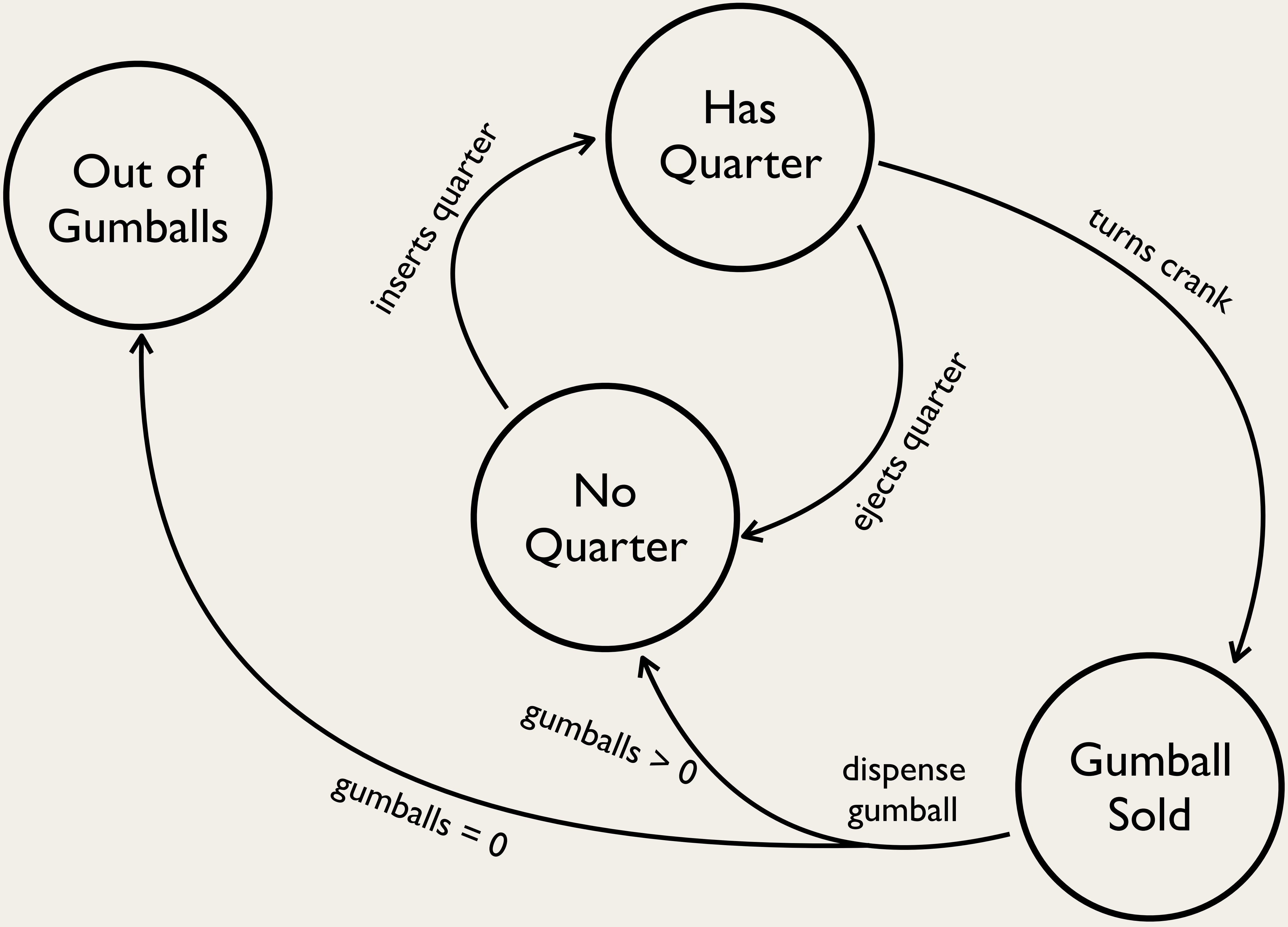
State machine for a gumball machine:

States

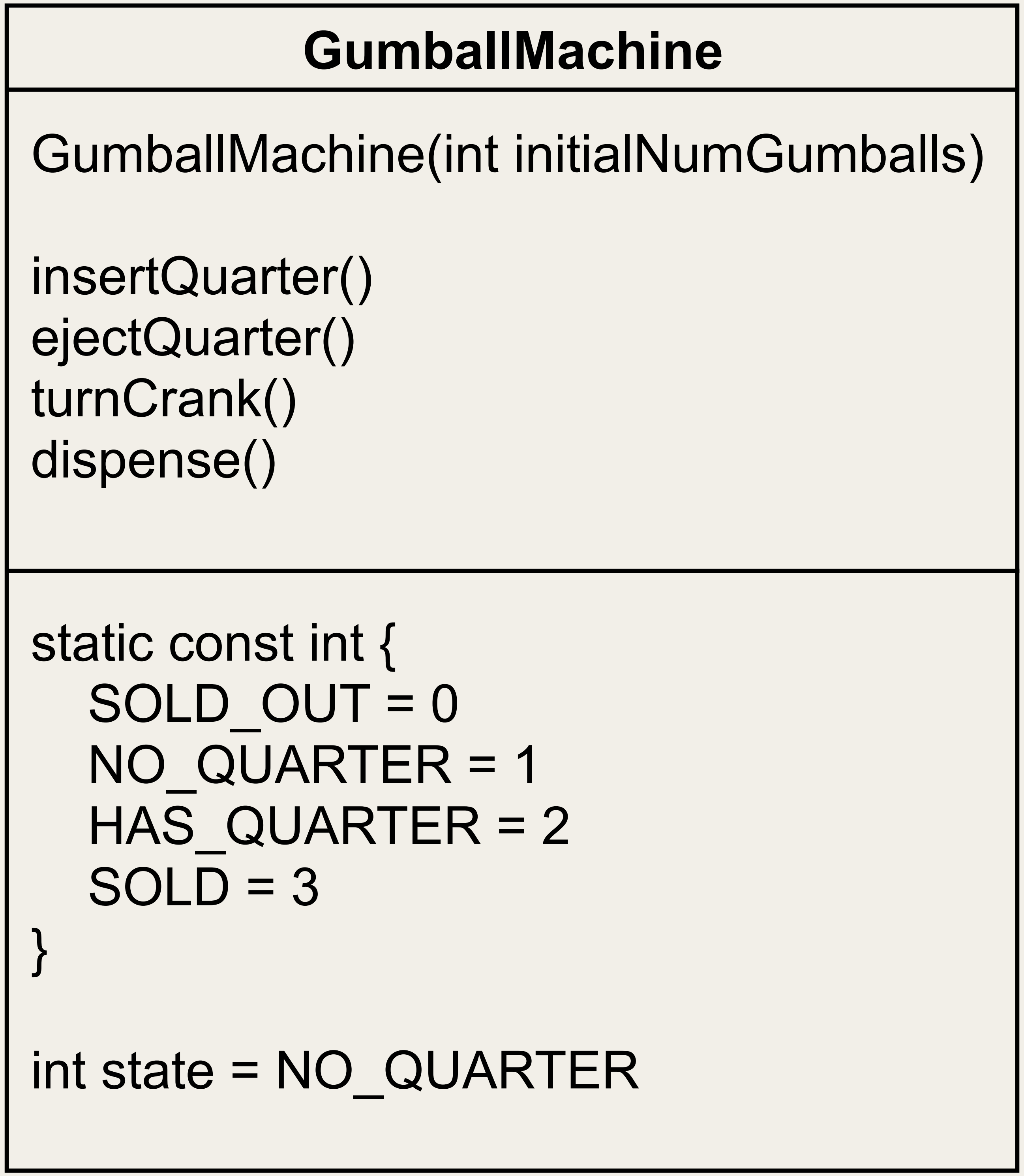
- No Quarter
- Has Quarter
- Gumball Sold
- Out of Gumballs

Transitions

- turns crank
- inserts quarter
- ejects quarter
- dispense gumball
  - gumballs > 0
  - gumballs = 0


Out of Gumballs
Has Quarter
inserts quarter
turns crank
ejects quarter
No Quarter
gumballs > 0
dispense gumball
gumballs = 0
Gumball Sold

How would you implement this in C++?



GumballMachine

```
GumballMachine(int initialNumGumballs)

insertQuarter()
ejectQuarter()
turnCrank()
dispense()
```

```
static const int {
    SOLD_OUT = 0
    NO_QUARTER = 1
    HAS_QUARTER = 2
    SOLD = 3
}

int state = NO_QUARTER
```

- Let’s write the insertQuarter() method as an example.
- Use print statements to print out what is happening, e.g. “You inserted a quarter” or “You can’t insert another quarter.”

GumballMachine::insertQuarter()

```
void GumballMachine::insertQuarter() {
   if (state == HAS_QUARTER) {
      cout << “You can’t insert another quarter” << endl;
   }
   else if (state == NO_QUARTER) {
      state = HAS_QUARTER;
      cout << “You inserted a quarter” << endl;
   }
   else if (state == SOLD_OUT) {
      cout << “You can’t insert a quarter, the machine is sold out” << endl;
   }
   else if (state == SOLD) {
      cout << “Please wait, we’re already giving you a gumball” << endl;
   }
}
```

Our First Implementation

- We defined the states as static constant integer values, with a member variable to hold the value of the current state.

```
static const int SOLD_OUT = 0;
static const int NO_QUARTER = 1;
static const int HAS_QUARTER = 2;
static const int SOLD = 3;
int state;
```

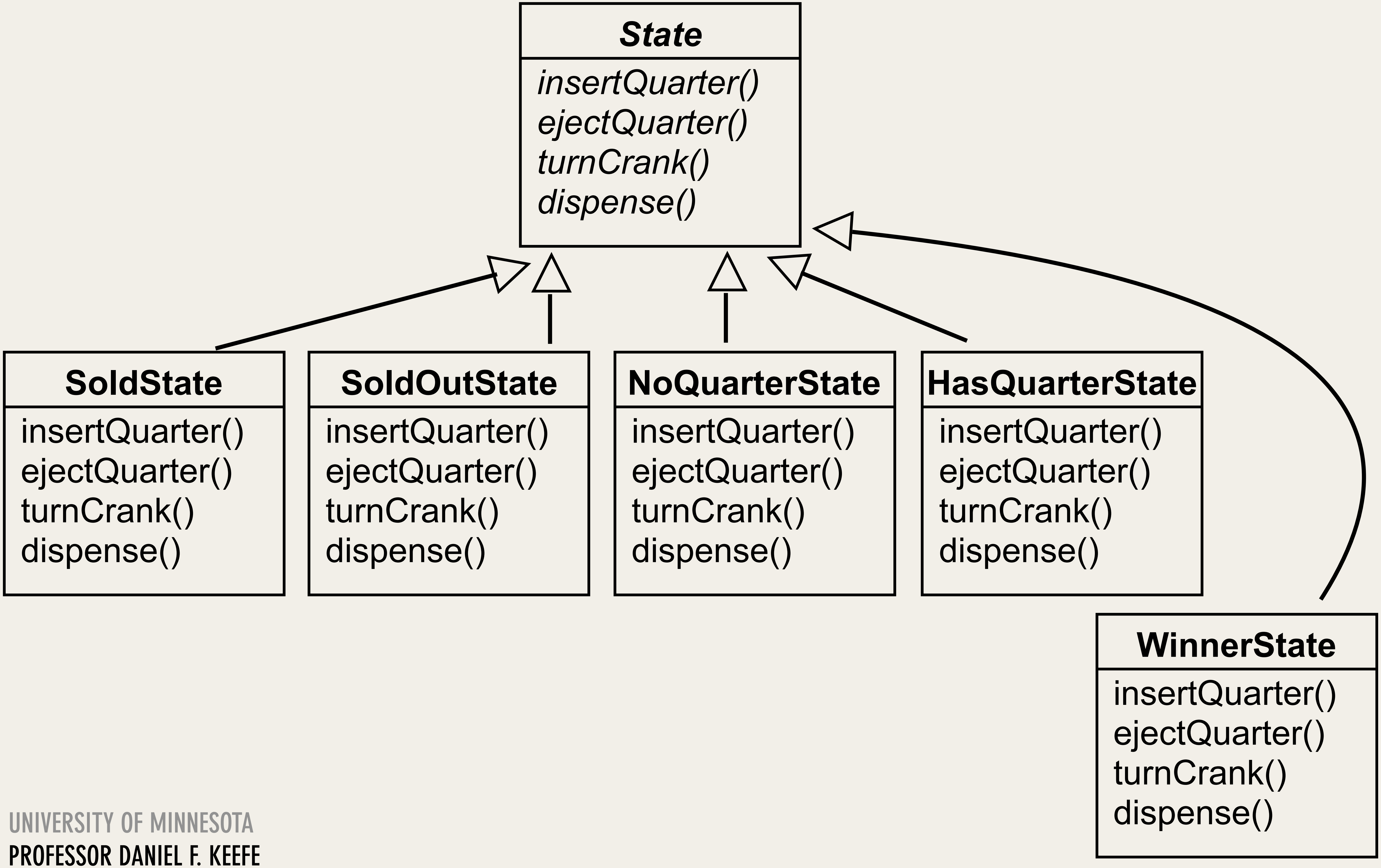
- Then, when some action occurs (e.g., insertQuarter) we use an if statement to determine the correct response based upon the current state.

```
if (state == NO_QUARTER) {
   state = HAS_QUARTER;
   cout << “You inserted a quarter” << endl;
}
```

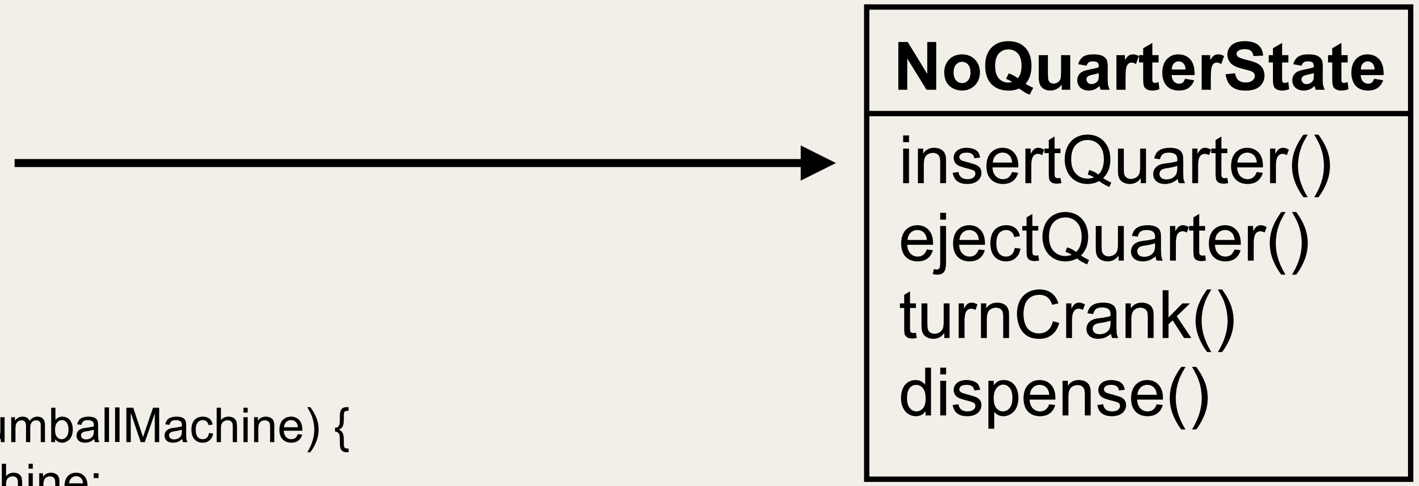
Our Usual Design Question: How Extensible is This?

- What if we add a new state: WINNER
- For every 1 of 10 gumballs sold, we assign a winner, and dispense an extra free gumball.
- What would we need to change?

An Alternative Design



Go to HasQuarterState

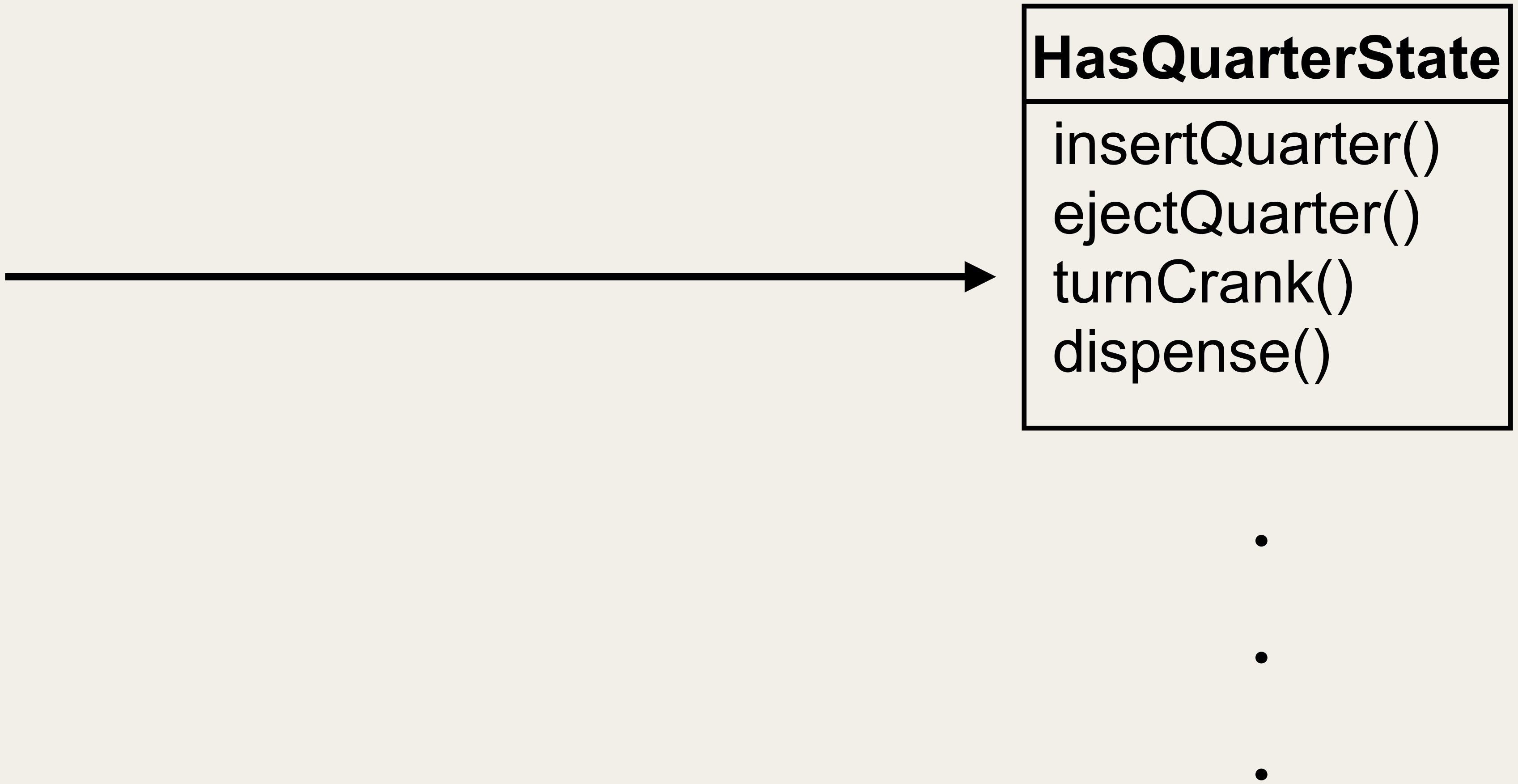


NoQuarterState

insertQuarter()
ejectQuarter()
turnCrank()
dispense()

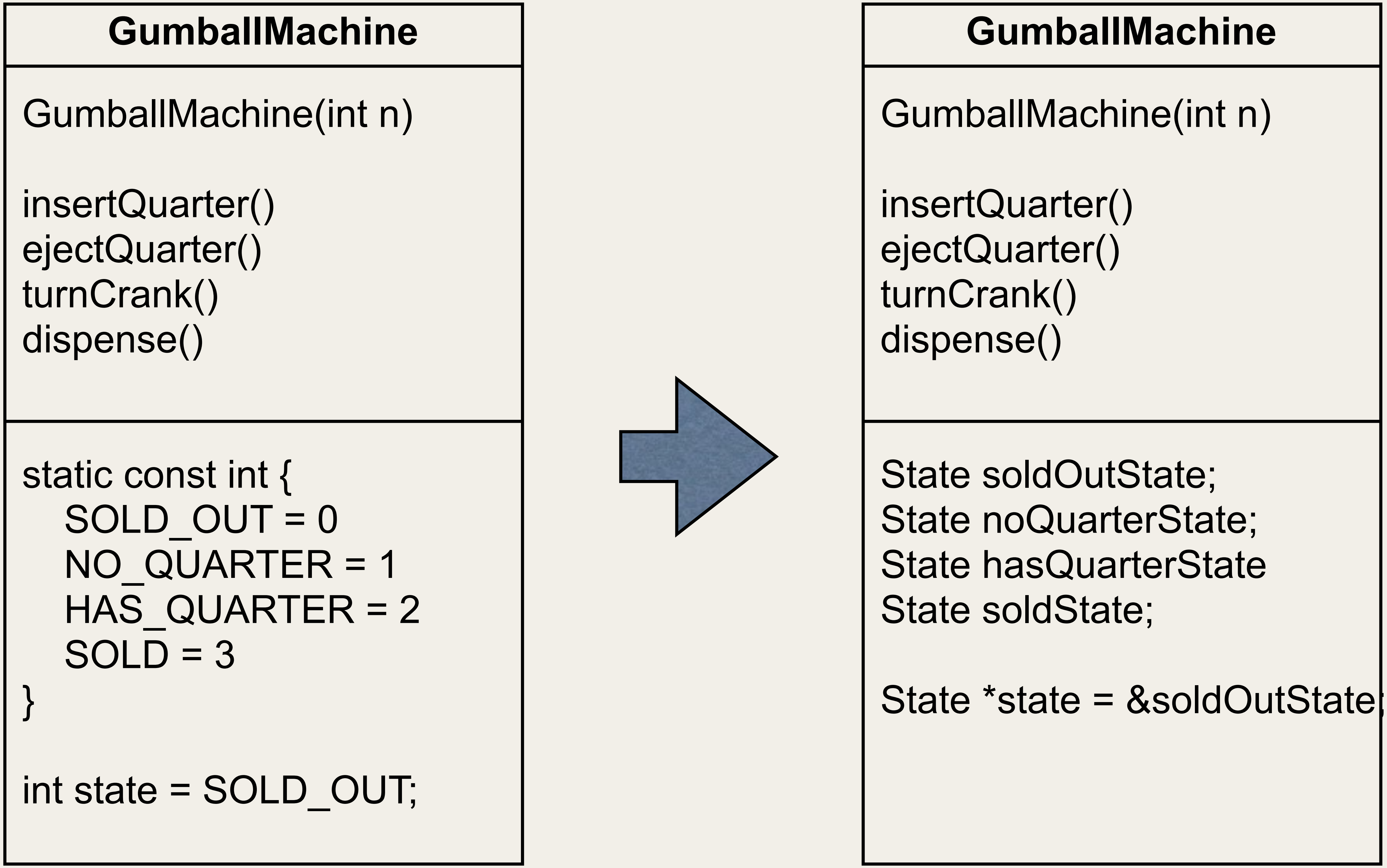
```
class NoQuarterState : public State {
public:
    NoQuarterState(GumballMachine &gumballMachine) {
        m_gumballMachine = gumballMachine;
    }

    void insertQuarter() {
        m_gumballMachine.setState(m_gumballMachine.getHasQuarterState());
    }

    …
};
```

Go to SoldState



HasQuarterState

insertQuarter()
ejectQuarter()
turnCrank()
dispense()



GumballMachine

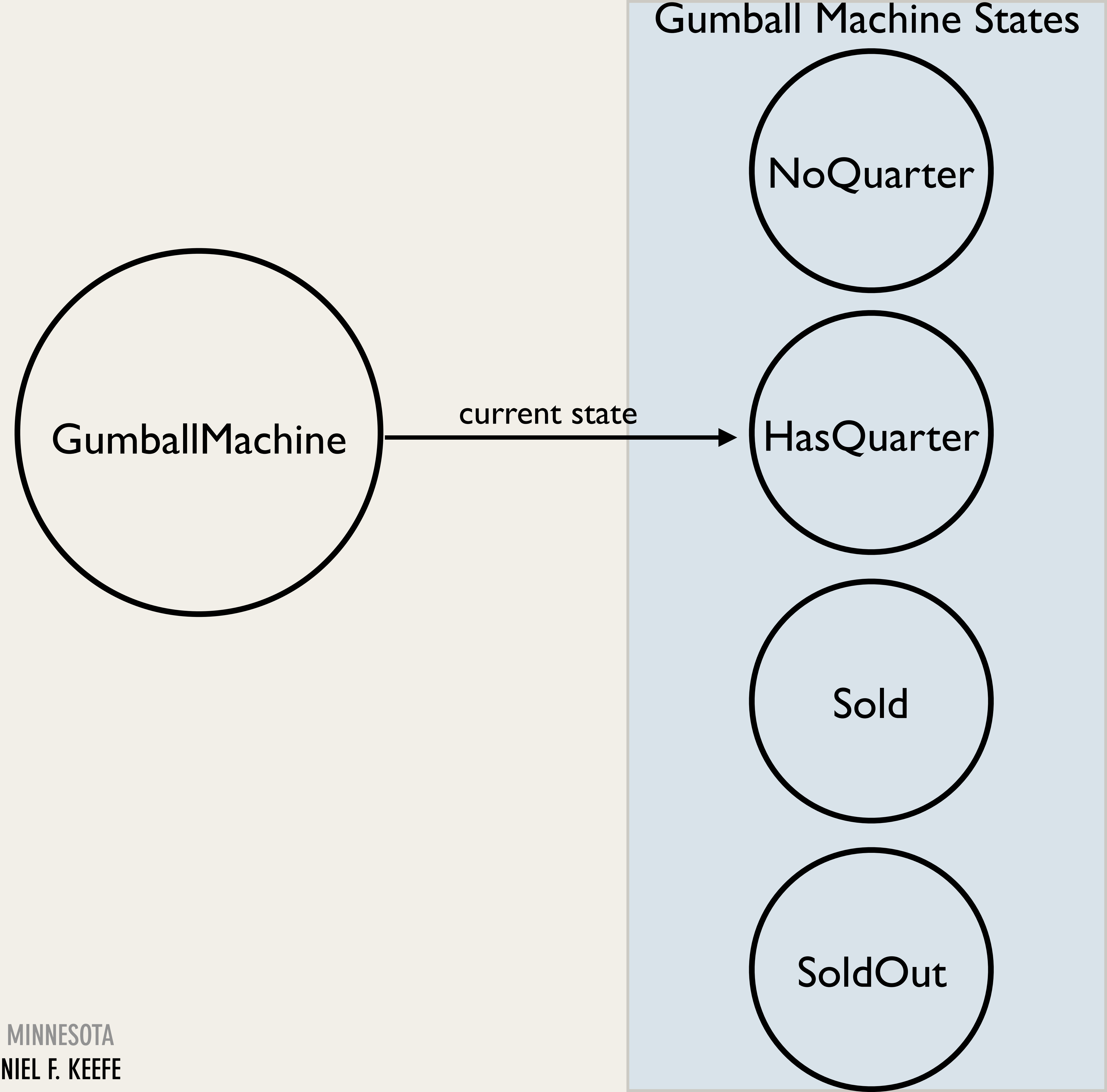
```
GumballMachine(int n)

insertQuarter()
ejectQuarter()
turnCrank()
dispense()
```

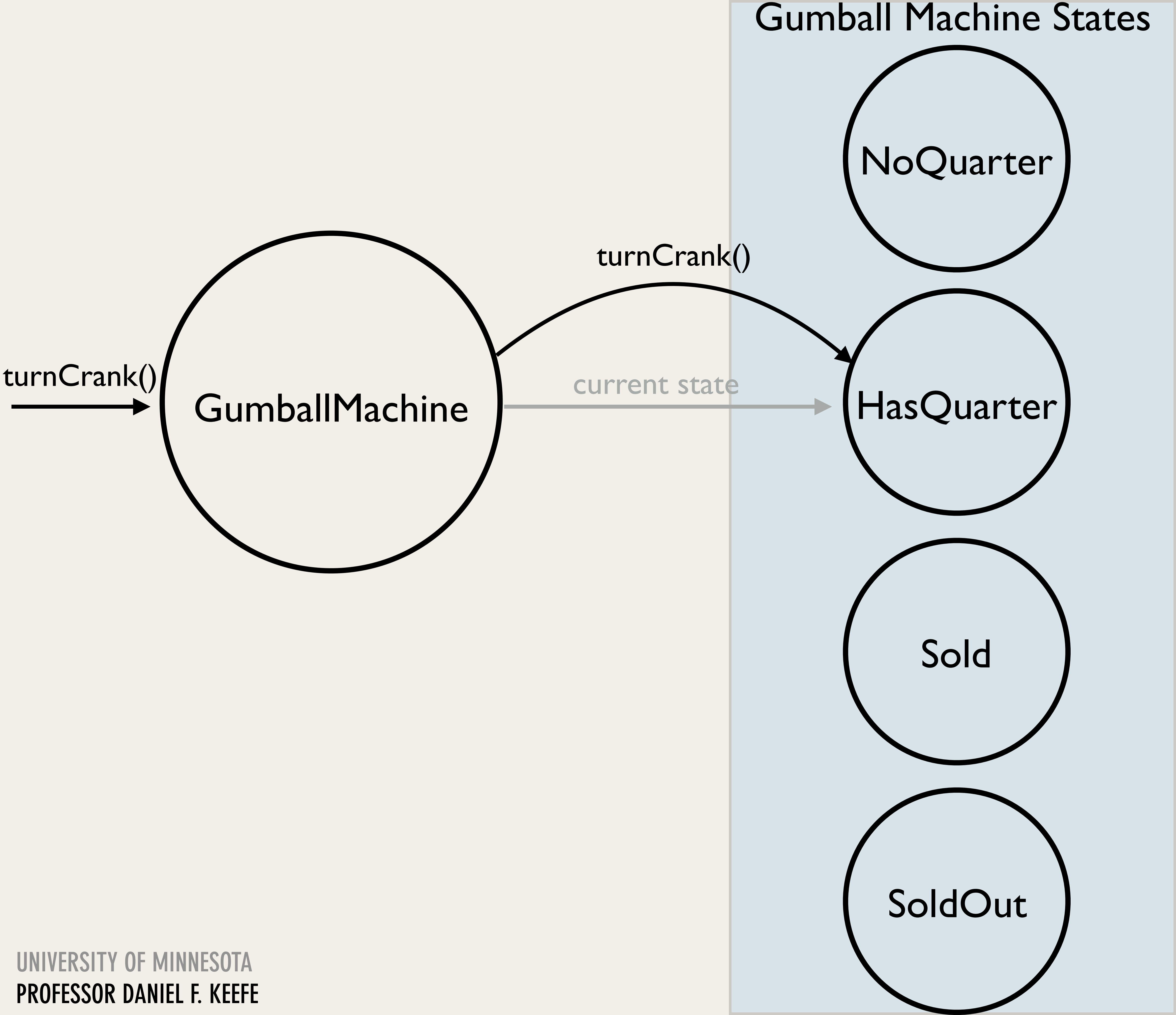
```
static const int {
   SOLD_OUT = 0
   NO_QUARTER = 1
   HAS_QUARTER = 2
   SOLD = 3
}

int state = SOLD_OUT;
```

GumballMachine

```
GumballMachine(int n)

insertQuarter()
ejectQuarter()
turnCrank()
dispense()
```

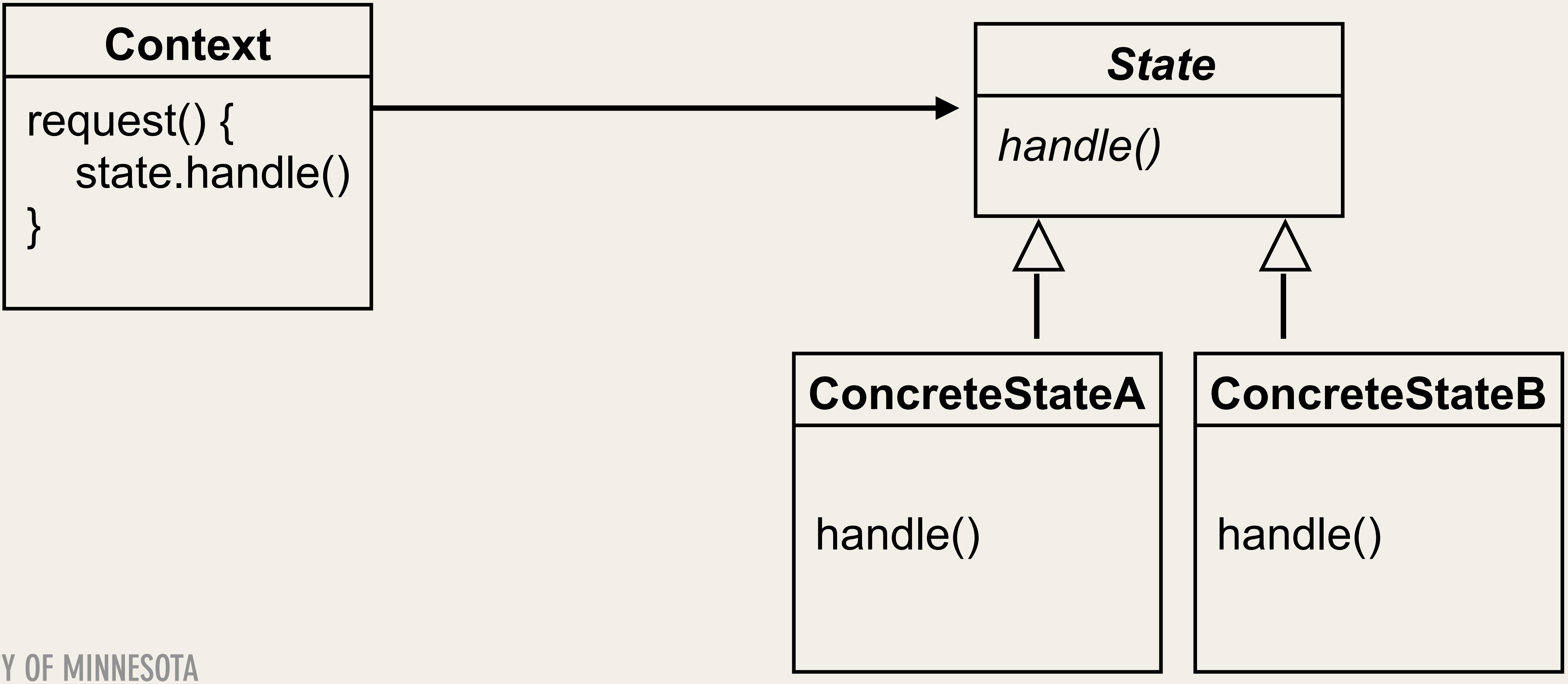
```
State soldOutState;
State noQuarterState;
State hasQuarterState
State soldState;

State *state = &soldOutState;
```


Gumball Machine States
NoQuarter
GumballMachine
current state
HasQuarter
Sold
SoldOut


Gumball Machine States
NoQuarter
turnCrank()
turnCrank()
GumballMachine
current state
HasQuarter
Sold
SoldOut

The State Pattern

- **The State Pattern** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



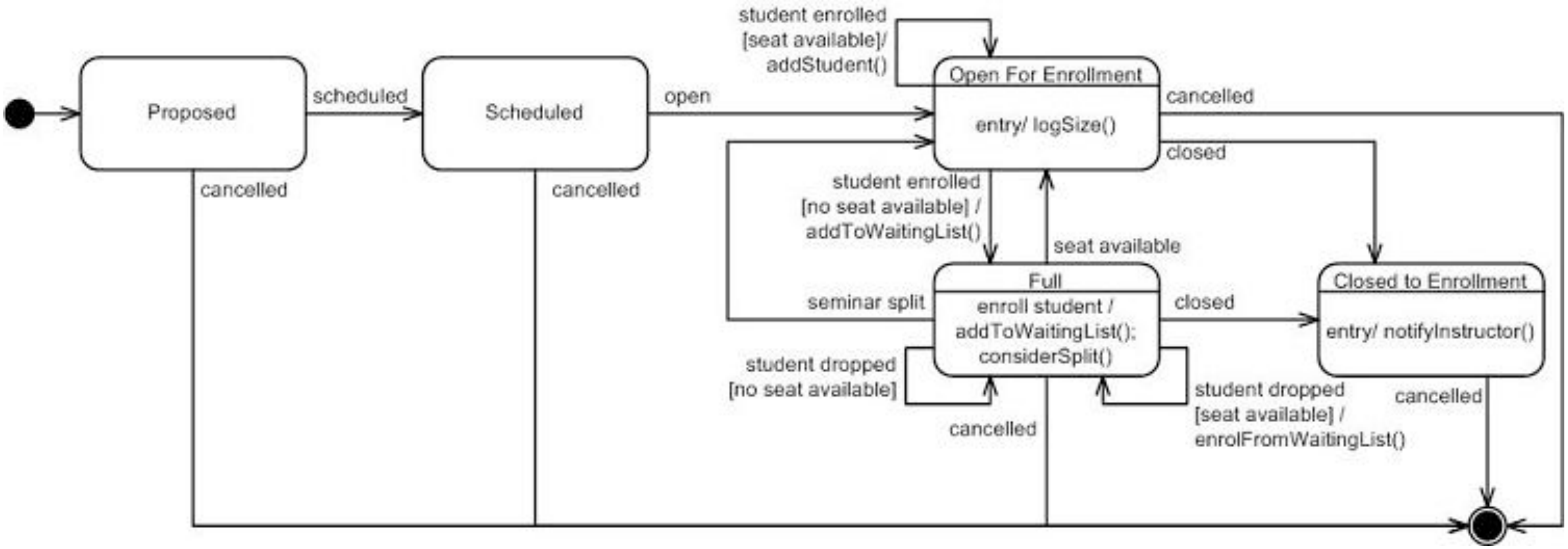
Design Discussion

- Let's take our project as an example…
- Is there a situation where the State Pattern might make sense?

UML State Diagrams

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

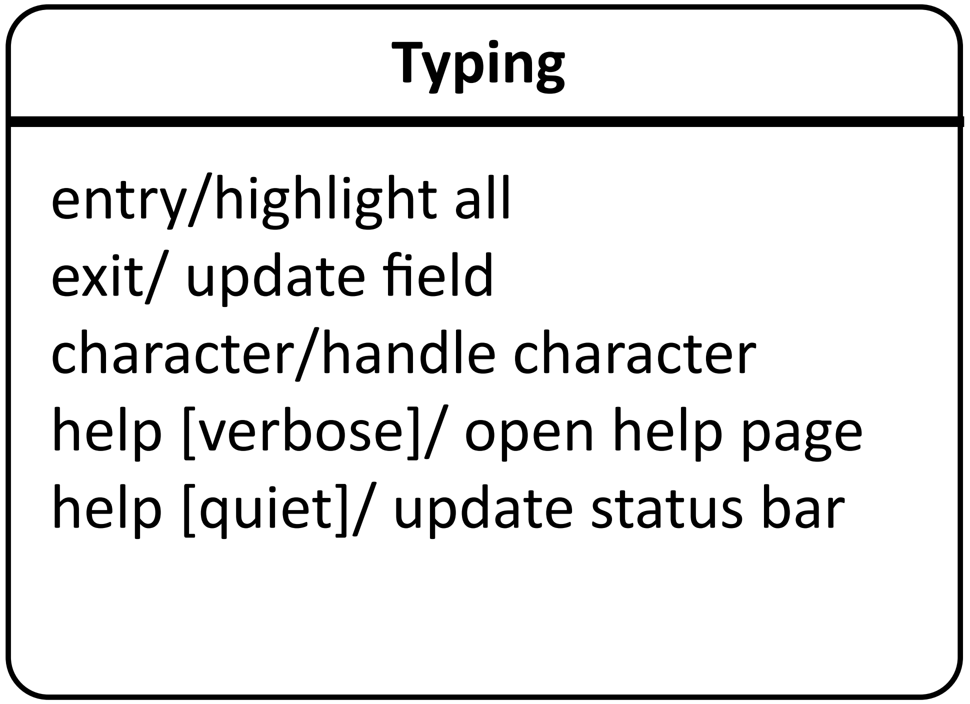
State Machine Diagrams



student enrolled [seat available]/ addStudent()
Proposed
scheduled
Scheduled
open
Open For Enrollment
entry/ logSize()
cancelled
closed
cancelled
cancelled
student enrolled [no seat available] / addToWaitingList()
seat available
Full
enroll student / addToWaitingList(); considerSplit()
seminar split
closed
Closed to Enrollment
entry/ notifyInstructor()
student dropped [no seat available]
cancelled
student dropped [seat available] / enrolFromWaitingList()
cancelled

- Components:
  - states
  - transitions -- trigger-signature [guard]/activity
  - initial pseudostate / final state

Internal Activities

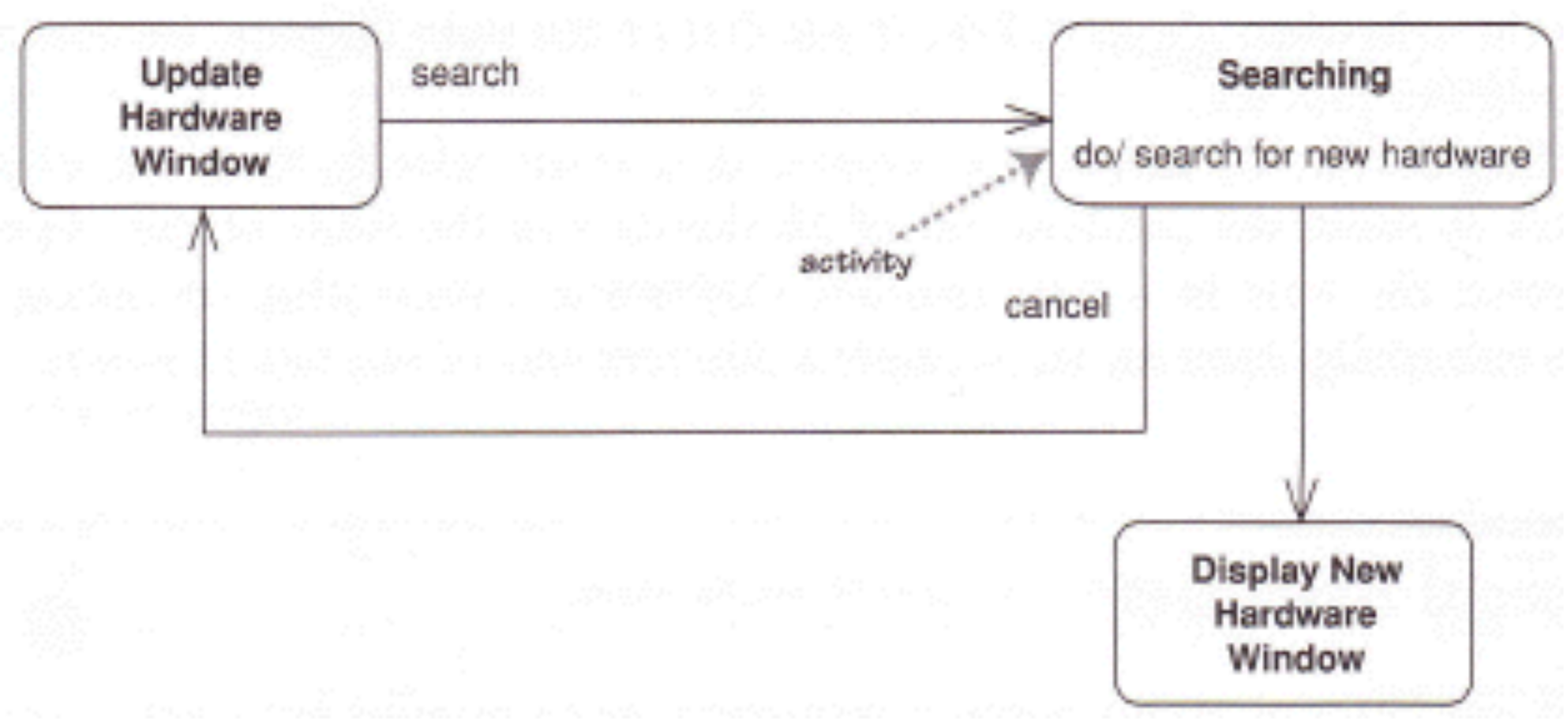


Typing

entry/highlight all
exit/ update field
character/handle character
help [verbose]/ open help page
help [quiet]/ update status bar

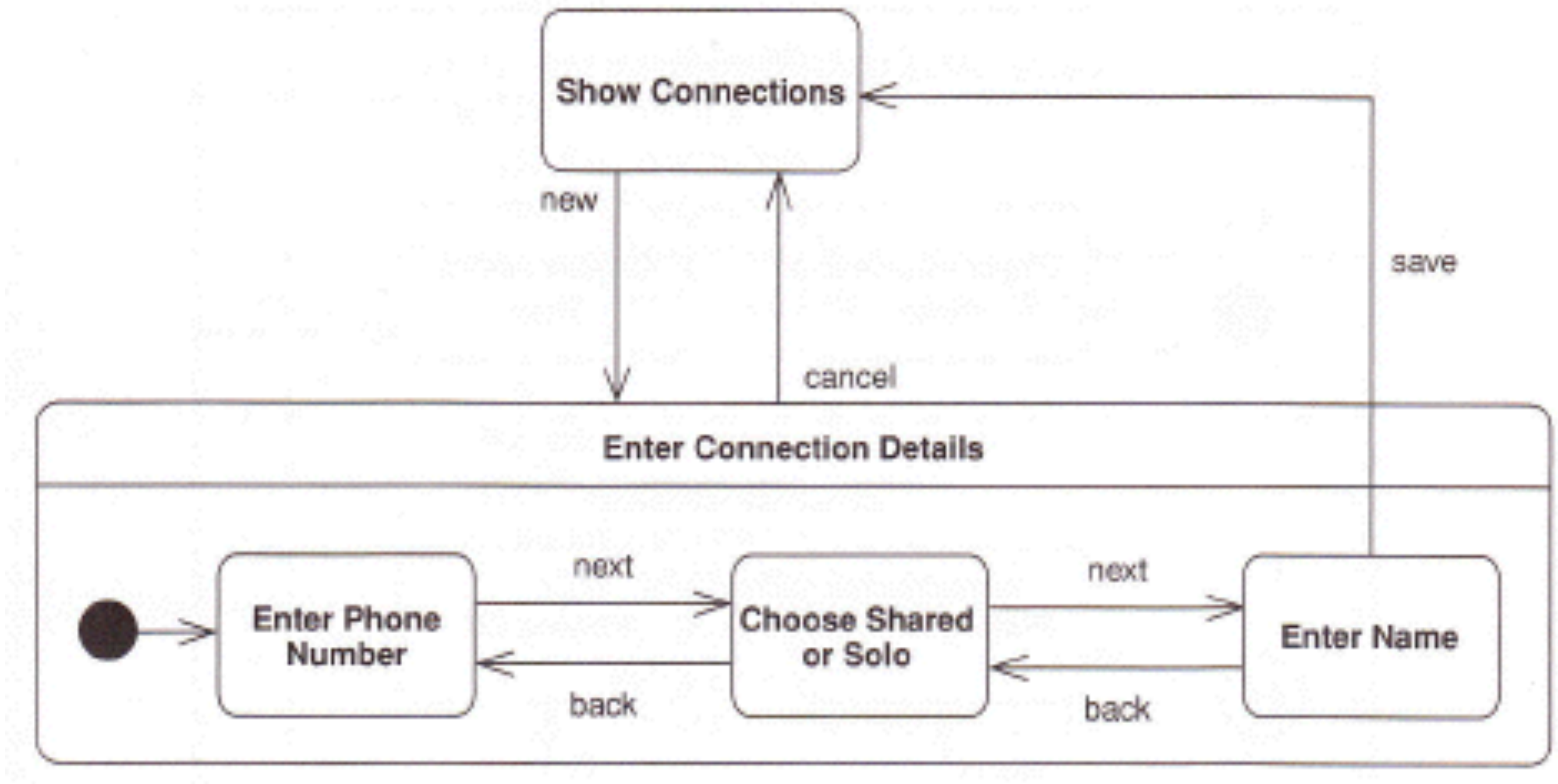
- States can react to events without transition
- Special entry and exit activities
- Similar to a self-transition, but internal activities do not trigger entry and exit activities, while self-transitions do.

Activities inside States



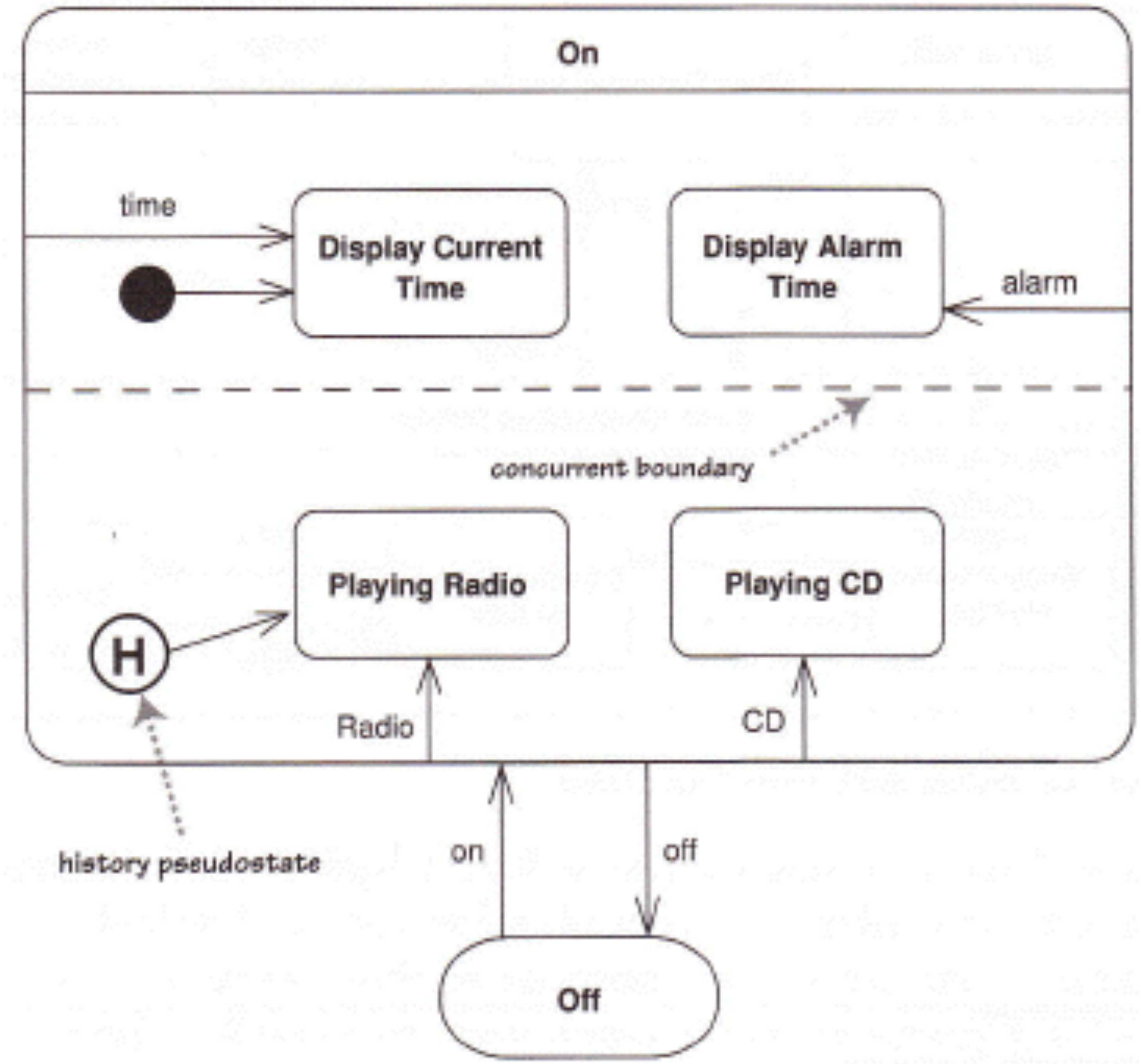
- Can have states in which the object does some ongoing activity, denoted by do/...
- When the activity ends, the transition without an event is taken.
- The “do” activity can be interrupted by an event (e.g. cancel).

Supersates



- Create a superstate when a group of states share common transitions and internal activities.

Concurrent Orthogonal States

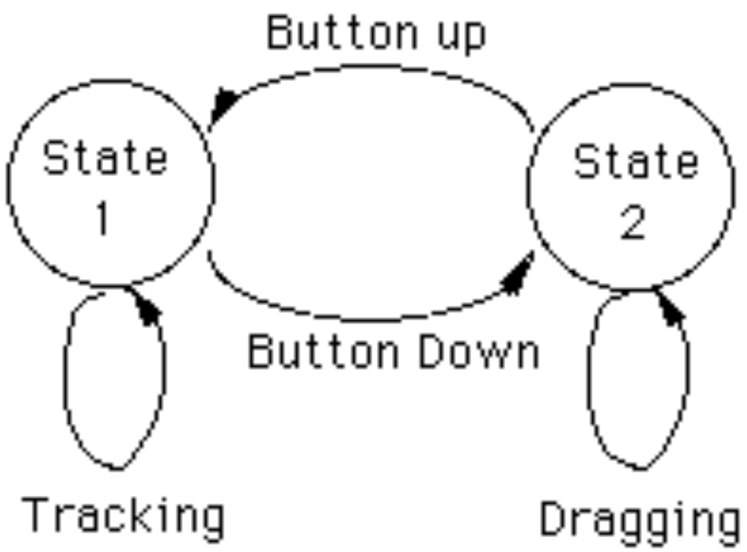


- CD/radio and current time/alarm time are orthogonal, so easiest to represent as separate state diagrams.

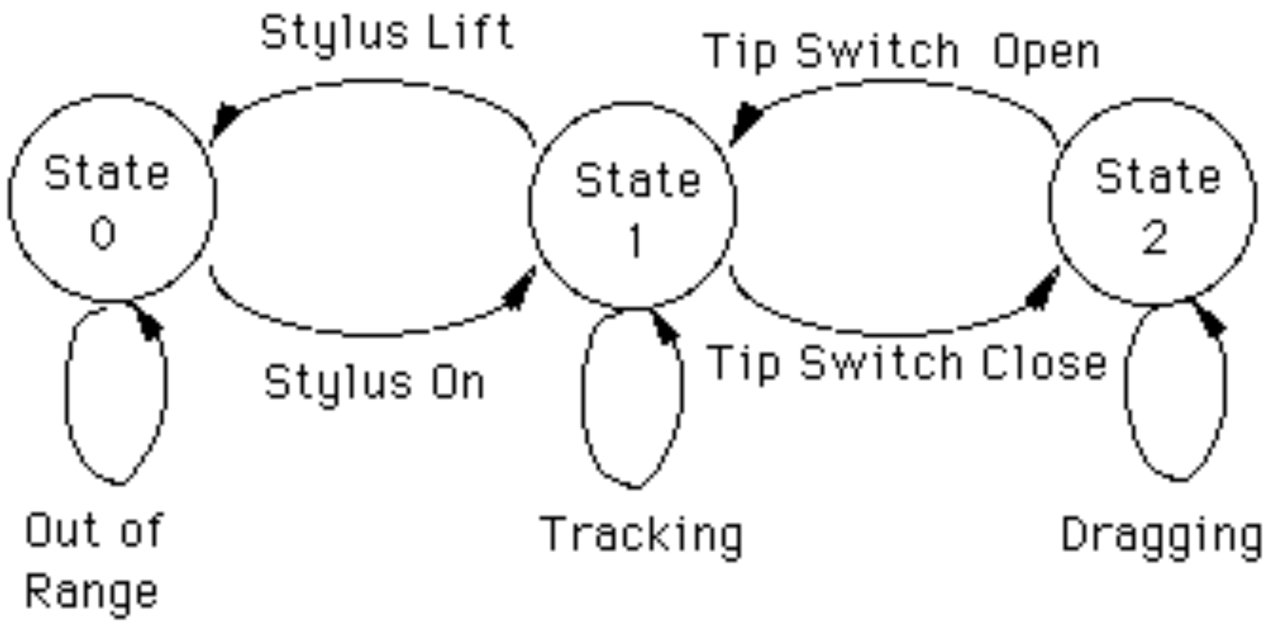
State Machines for User Input

A Three-State Model of Graphical Input

- Classic model by Bill Buxton: http://www.dgp.toronto.edu/OTP/papers/bill.buxton/3state.html
- Here's 2 of the 3 states... any idea what the third is?

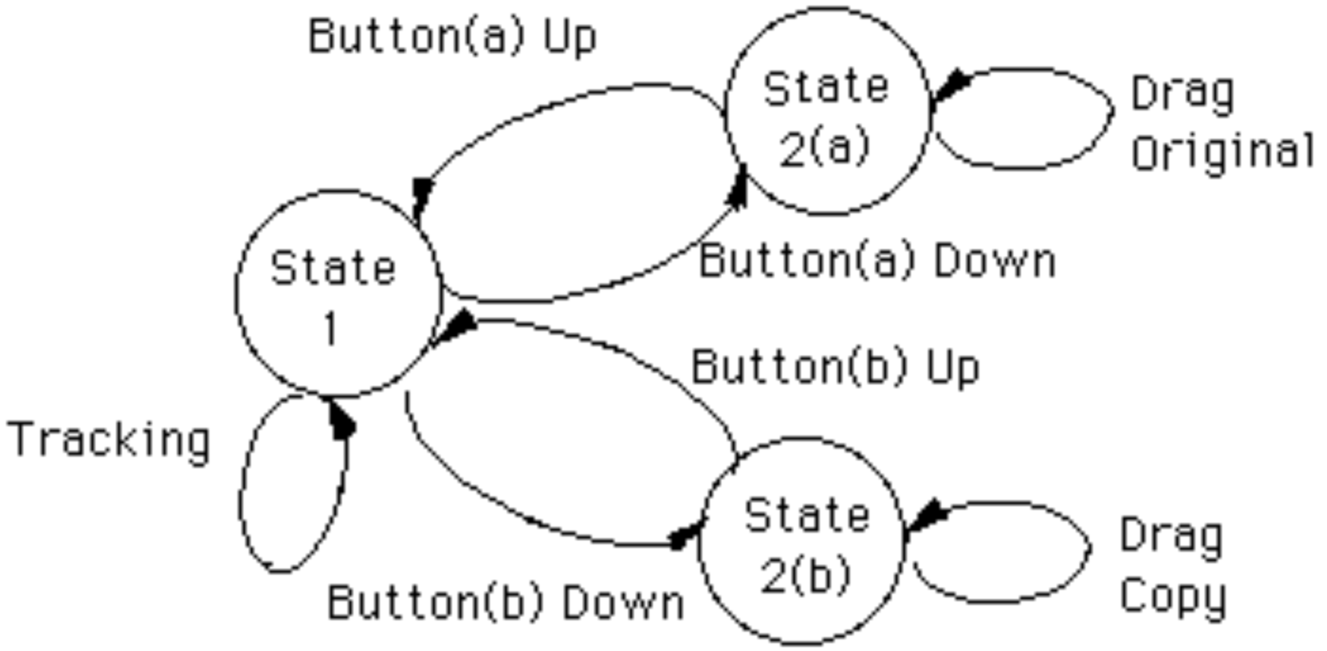


Buxton’s 3-state Model for Graphical Input



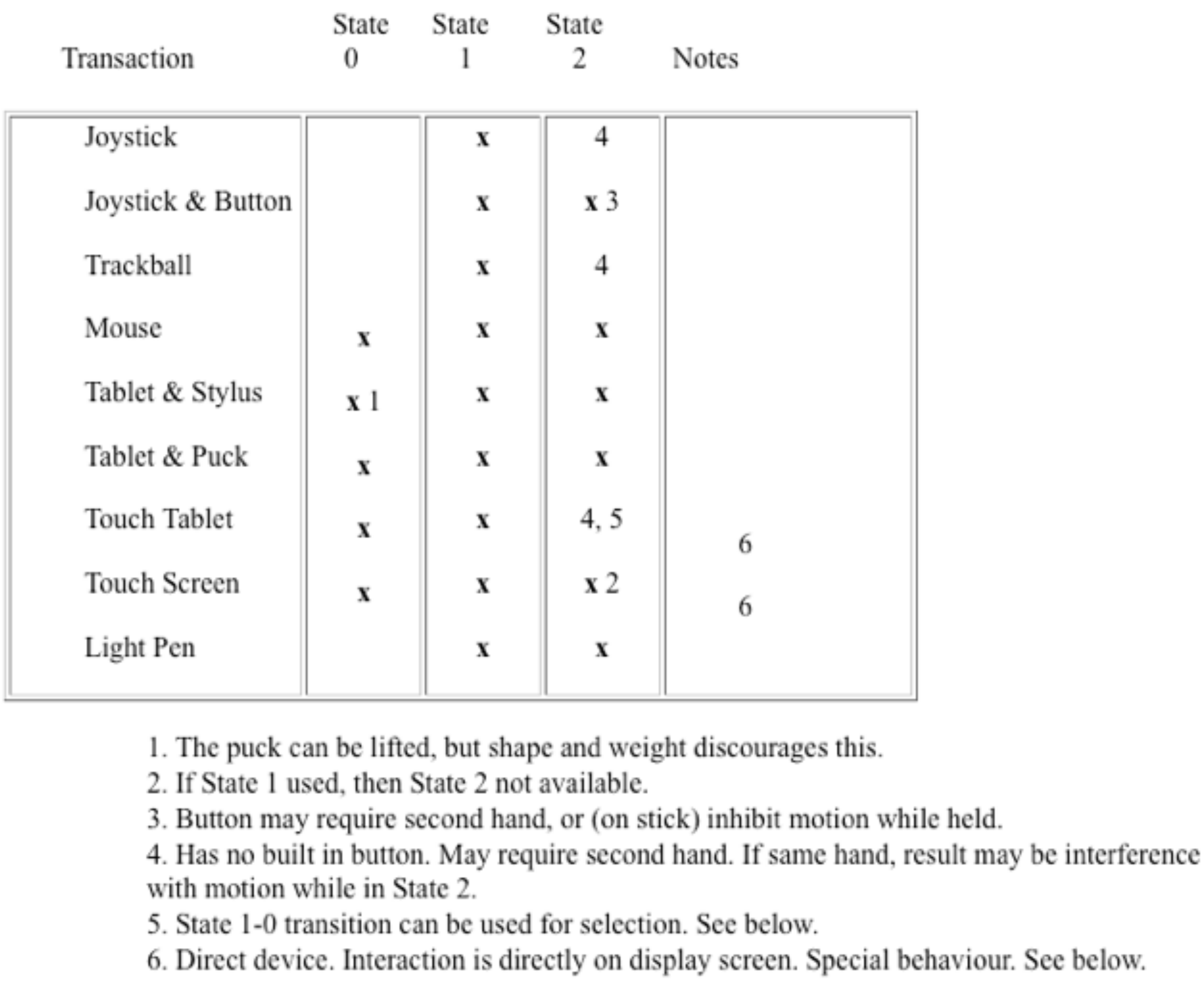
3 State Model (cont.)

- How would this change if you have a two button mouse?



3-State Model and Input Technologies

- Which states are available for various input devices:
- (Keep in mind, published in 1990.)



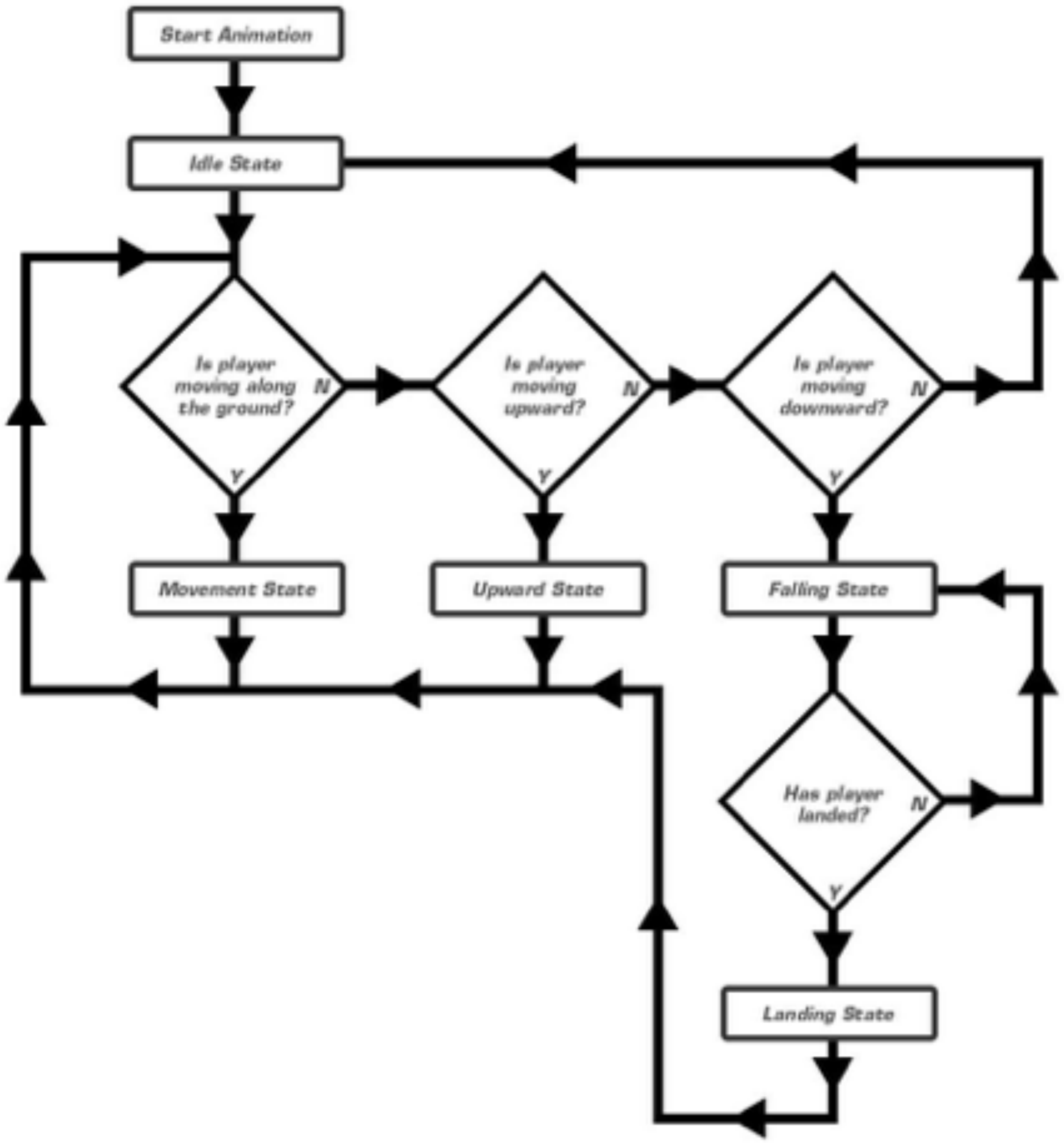
Transaction	State 0	State 1	State 2	Notes
Joystick		x	4	
Joystick & Button		x	x 3	
Trackball		x	4	
Mouse	x	x	x	
Tablet & Stylus	x 1	x	x	
Tablet & Puck	x	x	x	
Touch Tablet	x	x	4, 5	
Touch Screen	x	x	x 2	6
Light Pen		x	x	6

1. The puck can be lifted, but shape and weight discourages this.
2. If State 1 used, then State 2 not available.
3. Button may require second hand, or (on stick) inhibit motion while held.
4. Has no built in button. May require second hand. If same hand, result may be interference with motion while in State 2.
5. State 1-0 transition can be used for selection. See below.
6. Direct device. Interaction is directly on display screen. Special behaviour. See below.

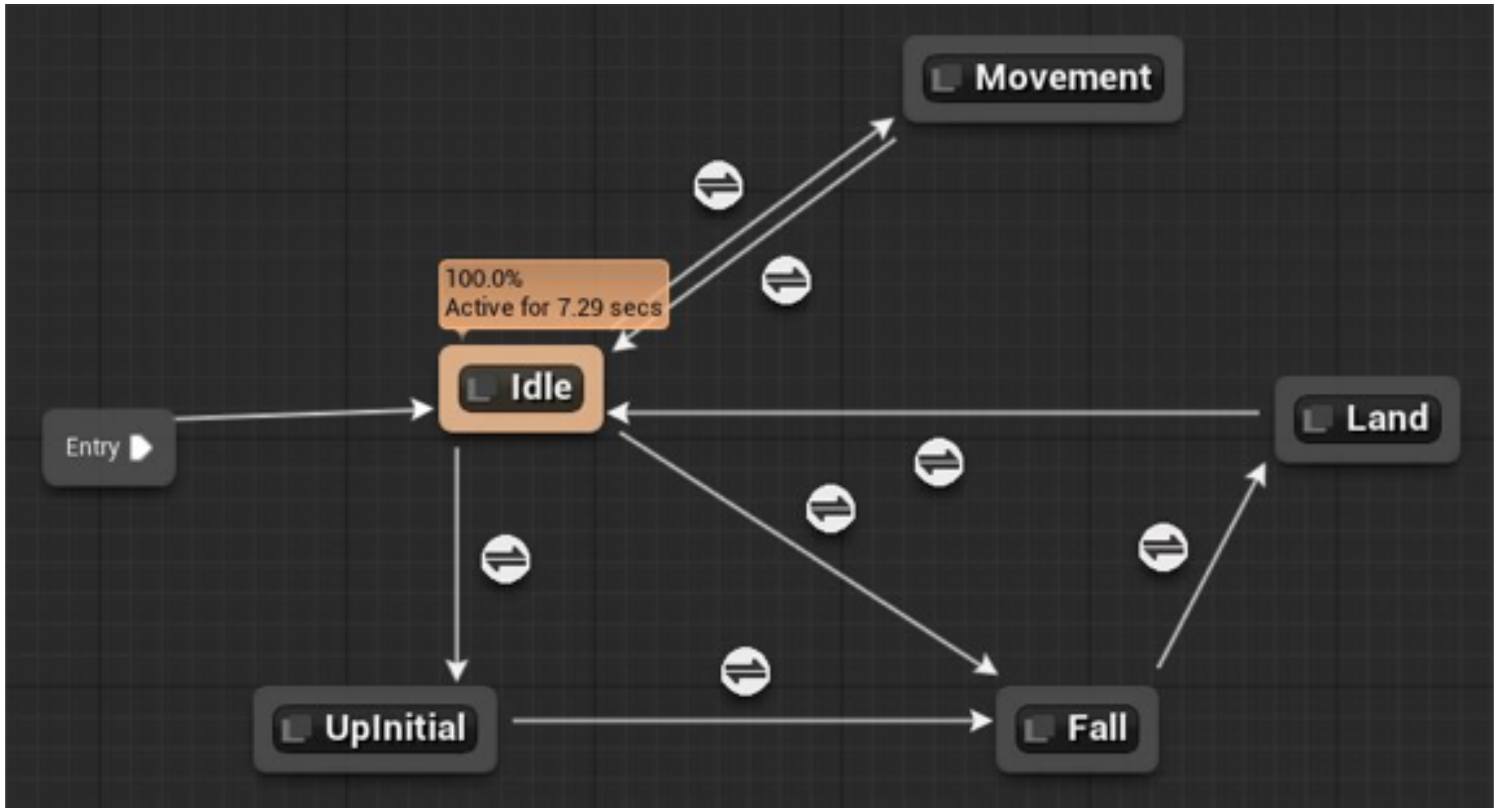
Other Cool Examples of State Machines

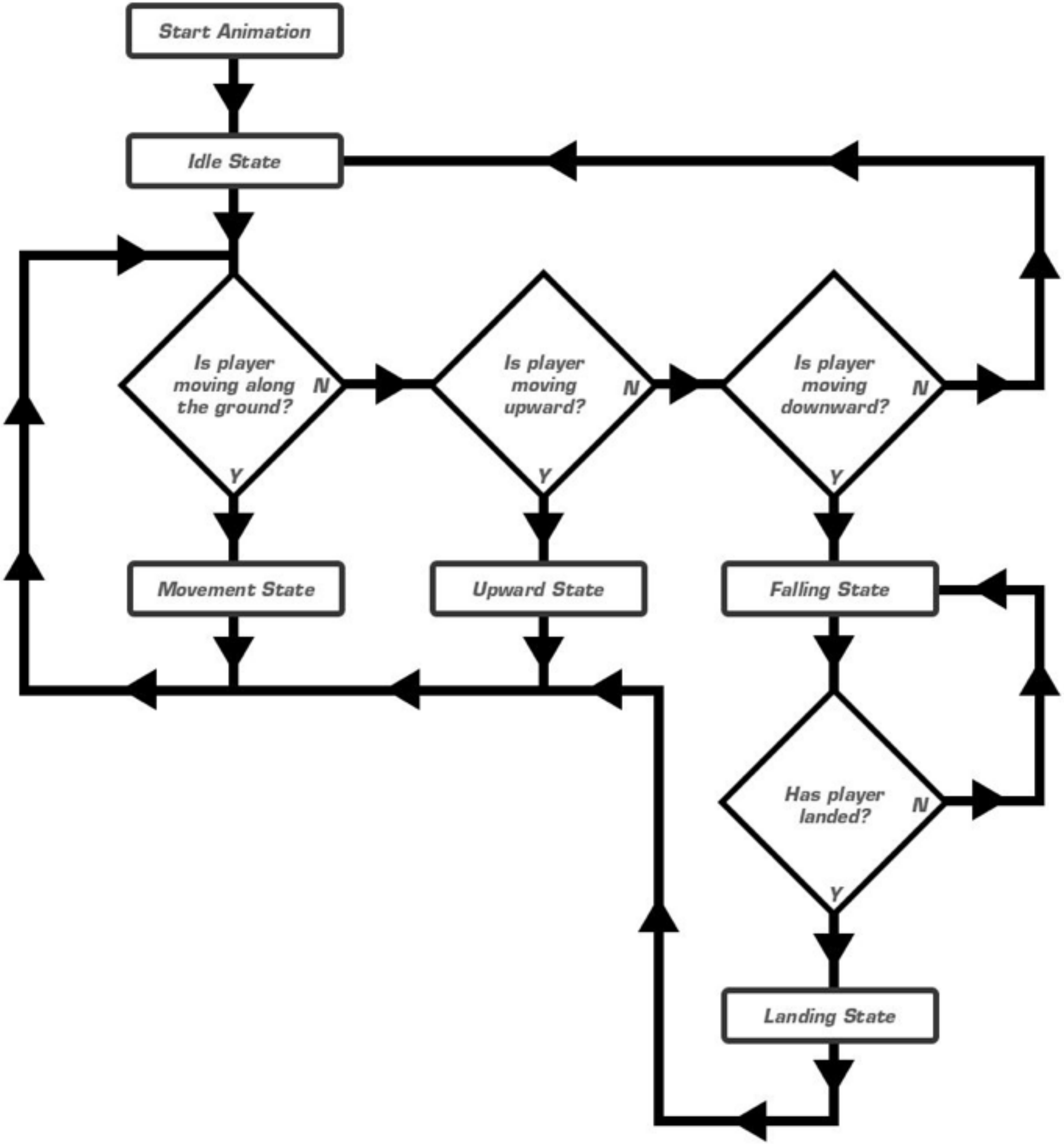
- Character animation with the Unreal Game Engine

Conceptual Flow Chart



State Machine Editor





Start Animation

Idle State

Is player moving along the ground?

N

Y

Is player moving upward?

N

Y

Is player moving downward?

N

Y

Movement State

Upward State

Falling State

Has player landed?

N

Y

Landing State