Review and Wrap Up

CSci-3081W: Program Design and Development

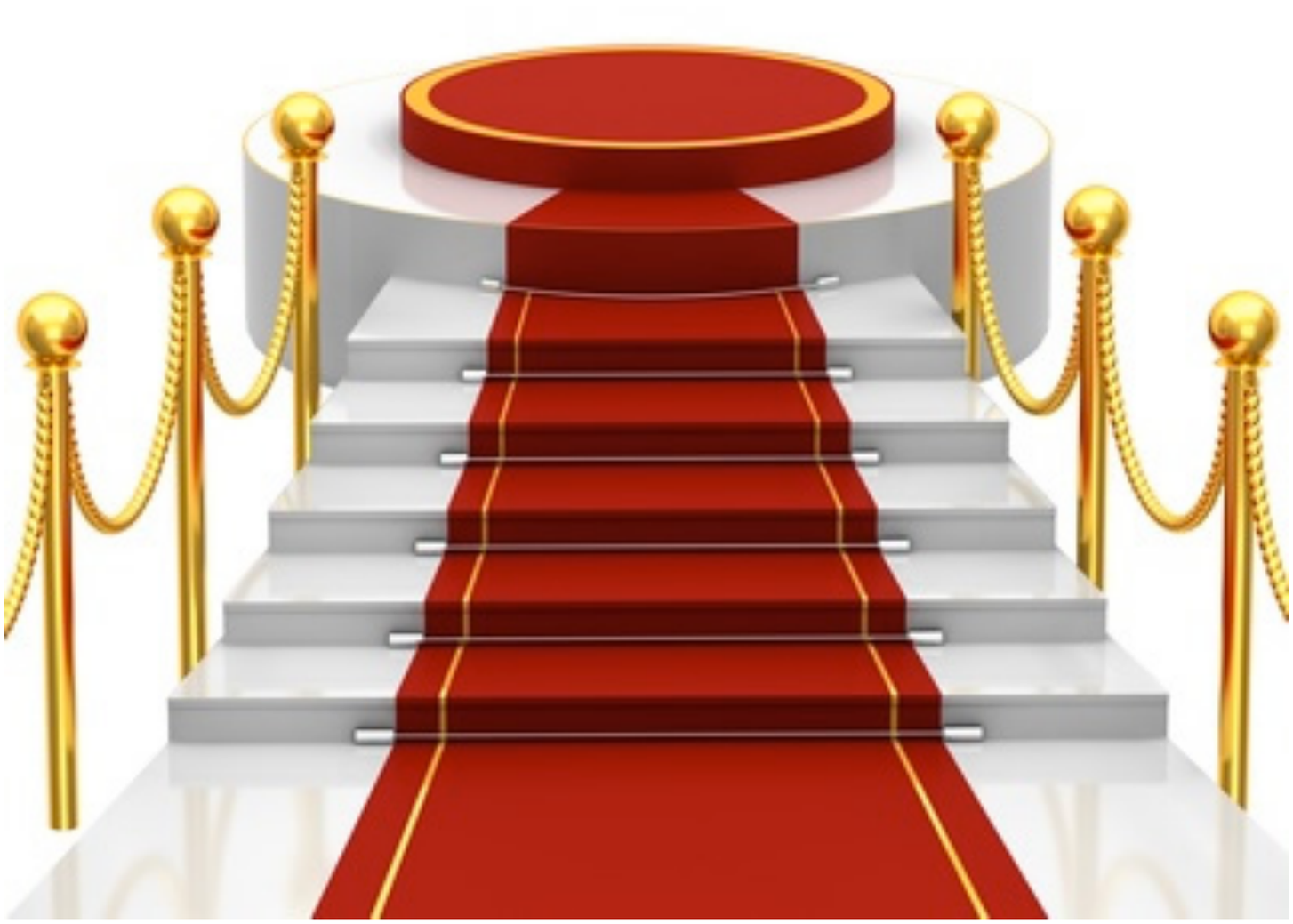
Final Exam Timing and Typical Format

- Sat 5/14, 1:30pm - 3:30pm, Here.
- Example format from last year's exam:
    - 10 short answer questions
        - write a few lines of code
        - answer in 1-2 sentences
    - 3 longer answer questions
        - UML diagramming
        - mini-essay (2-3 paragraphs)
- Closed book exam, but you may use 1 page (front and back) of ***Hand-Written-By-You*** notes.

Today: The Semester in a Nutshell

- We've come a long way in this semester. Remember, starting off with class diagrams (Ducks) and compiling.
- Today, let's discuss one or two slides from each class session that I think belong in the 3081W slide "Hall of Fame".

CSci-3081W
Slide Hall of Fame



UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Big Topic #1: Software Design

Big Topic #2: Writing, Including Software Diagramming

Big Topic #3: C++ Software Development

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Big Topic #1: Software Design

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Benefits of Using ADTs (classes)

Let's explain these:

1. You can hide implementation details
2. Changes don't affect the whole program
3. You can make the interface more informative
4. It's easier to improve performance
5. The program is more obviously correct
6. The program becomes more self-documenting
7. You don't have to pass data all over your program
8. You're able to work with real-world entities rather than low-level implementation structures

C++ Example of a Class Interface with Mixed Levels of Abstraction

```
class EmployeeCensus: public ListContainer {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );

   Employee NextItemInList();
   Employee FirstItem();
   Employee LastItem();
   ...
private:
   ...
};
```

The abstraction of these routines is at the "employee" level.

The abstraction of these routines is at the "list" level.

- Poor abstraction, mixed levels of abstraction.
- Each class should implement one and only one abstract data type.

Client

Duck

performQuack()
performFly()
swim()
display()
// other duck-like methods…

MallardDuck

display() {
  // looks like a mallard
}

RedheadDuck

display() {
  // looks like a redhead
}

RubberDuck

display() {
  // looks like a rubber duck
}

DecoyDuck

display() {
  // looks like a decoy duck
}

Encapsulated fly behavior

flyBehavior

FlyBehavior

fly()

FlyWithWings

fly() {
  // implements duck flying
}

FlyNoWay

fly() {
  // do nothing - can't fly!
}

Encapsulated quack behavior

quackBehavior

QuackBehavior

quack()

Quack

quack() {
  // implements duck quacking
}

Squeak

quack() {
  // rubber ducky squeak
}

MuteQuack

quack() {
  // do nothing - can't quack
}

Think of these as a family of interchangeable algorithms!

Summary of Two Types of Factories

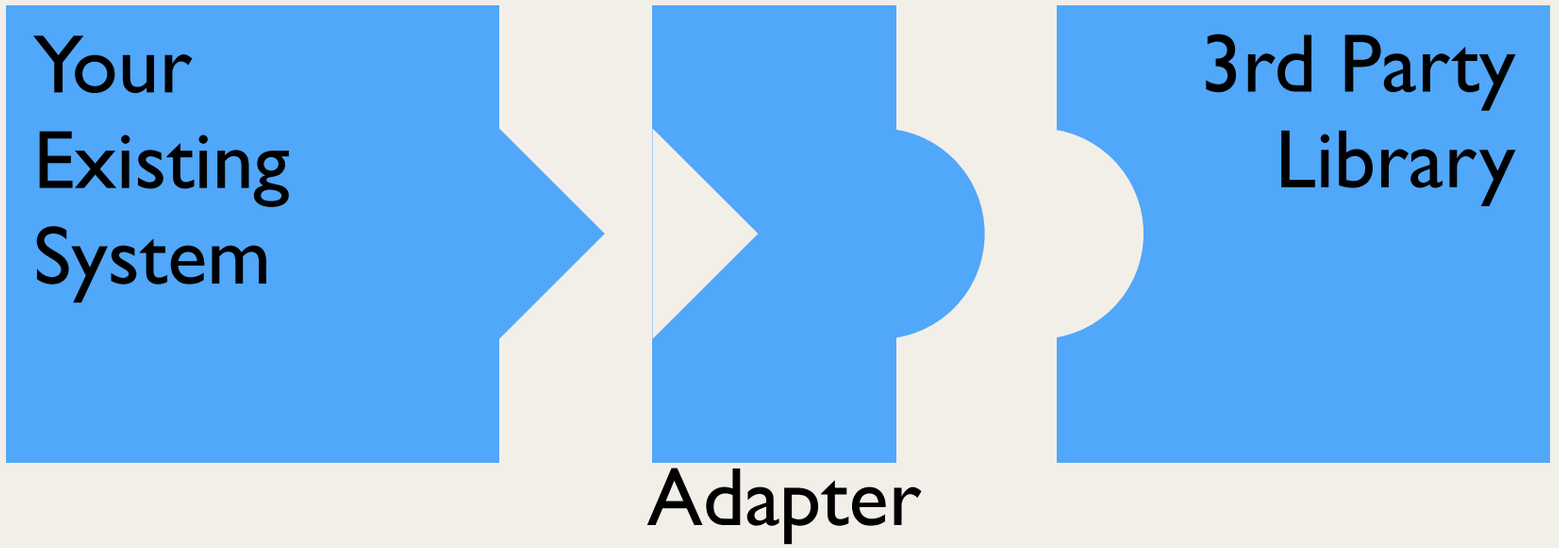
- **Simple Factory:**
  - You identify that the code that is changing a lot has to do with figuring out which type of object (e.g., Pizza) to create because you keep needing to add new Pizzas and take away old Pizzas.
  - Move all code that creates objects Pizzas to a single Factory class. This is the only place in your application where Pizzas are created.
  - This isolates change and encapsulates object creation, so this is a good programming design decision.
- **Factory Method:**
  - If we have more variation in our program, for example, multiple PizzaStores that each create their own versions of pizzas. Then, we need a bit more flexibility.
  - Using a factory method is a good choice here because it has the same advantages of the simple factory while also allowing us to vary the products we create based on the subclass we are in.

Meet the Template Method Design Pattern

```
class CaffeineBeverage {
public:
  void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
  }

  virtual void brew() = 0;

  virtual void addCondiments() = 0;

  void boilWater() {
    // implementation…
  }

  void pourInCup() {
    // implementation…
  }
};
```

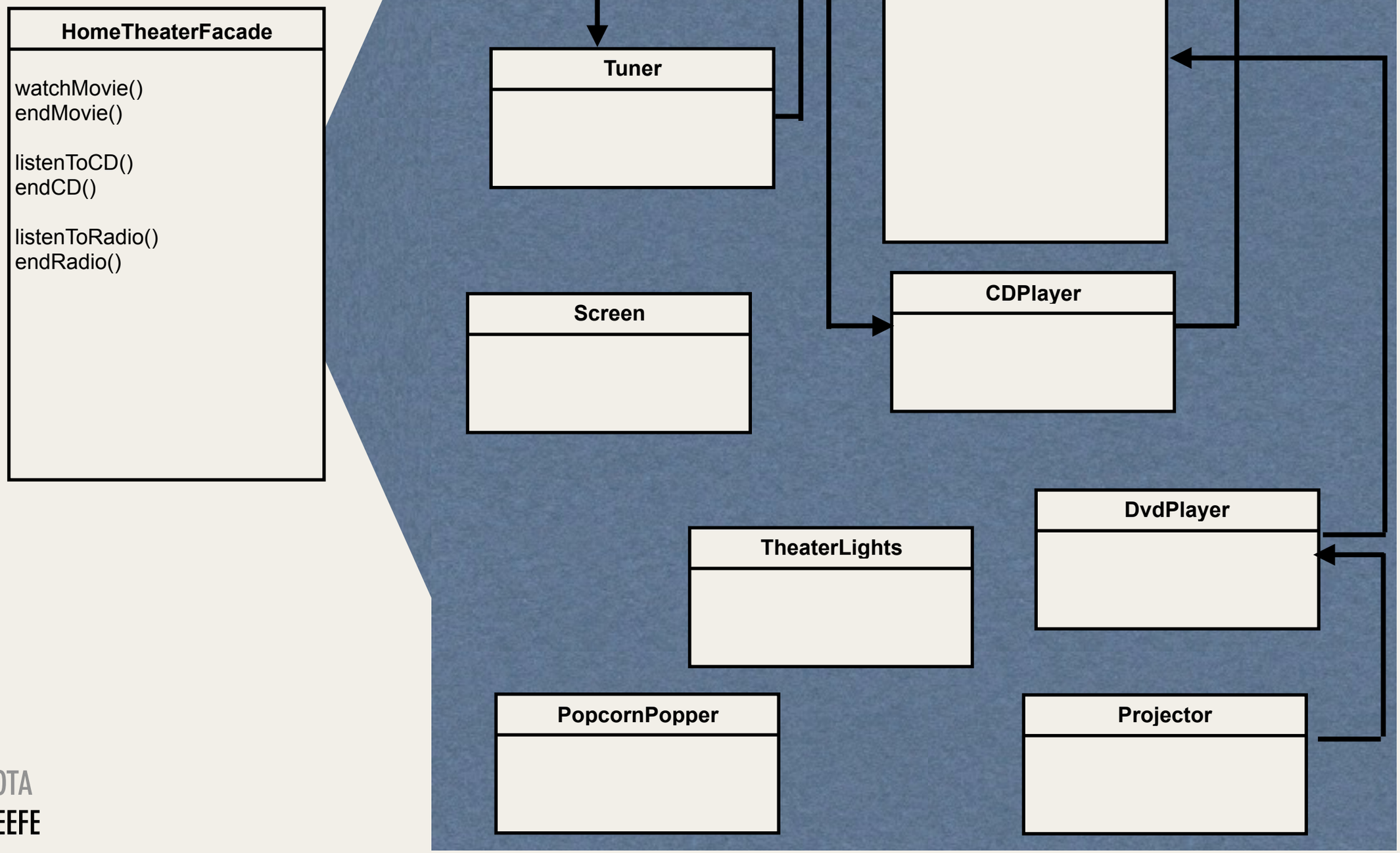
- prepareRecipe() is a template method. Why?
- Each step of the algorithm is represented as a method.
- Some methods are handled by this class.
- And other methods are handled by the subclass.

Object-Oriented Adapters



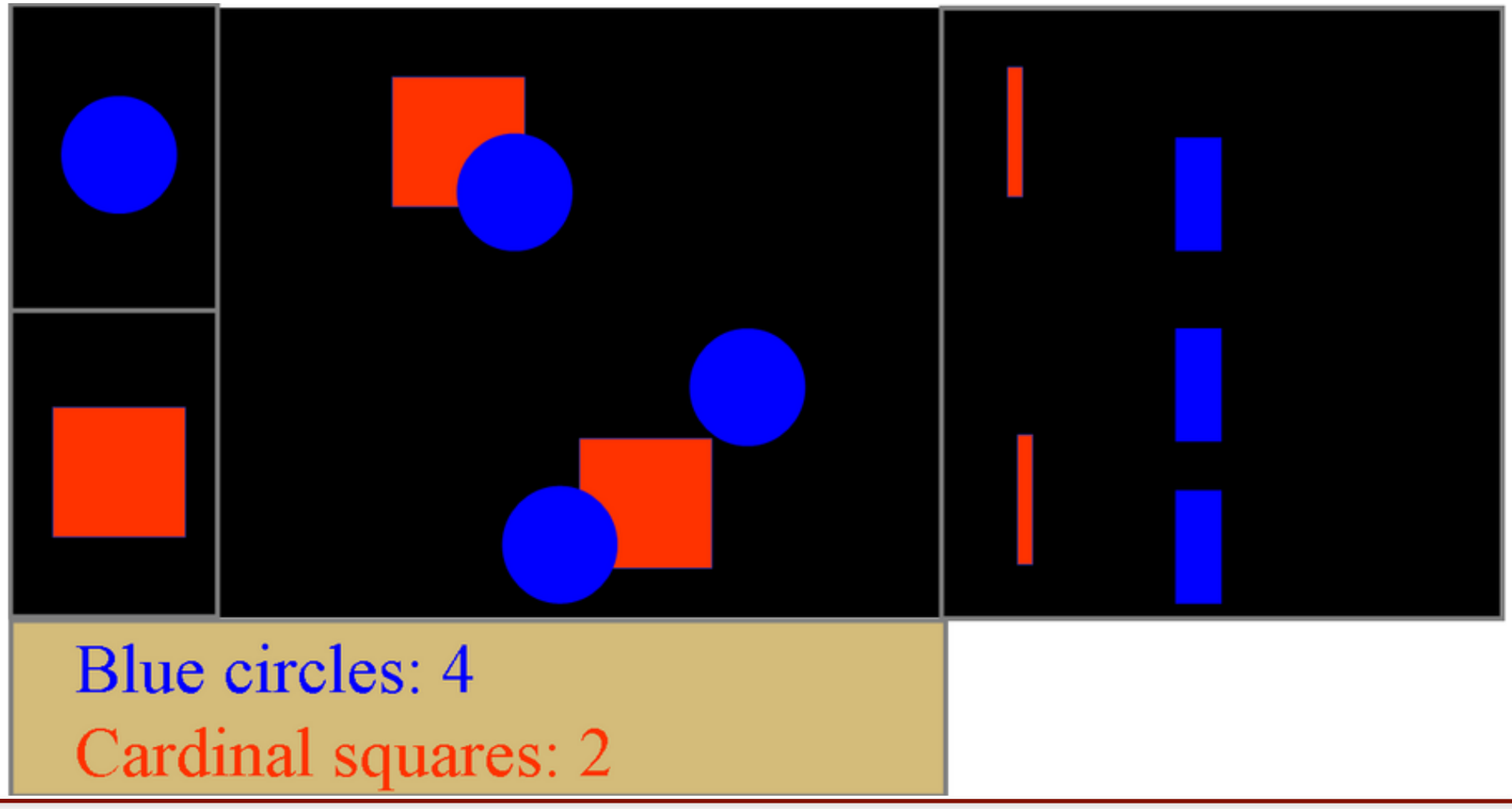
- You write new code for the Adapter.
- But, there should be no code changes for Your Existing System and no code changes for the 3rd Party Library.

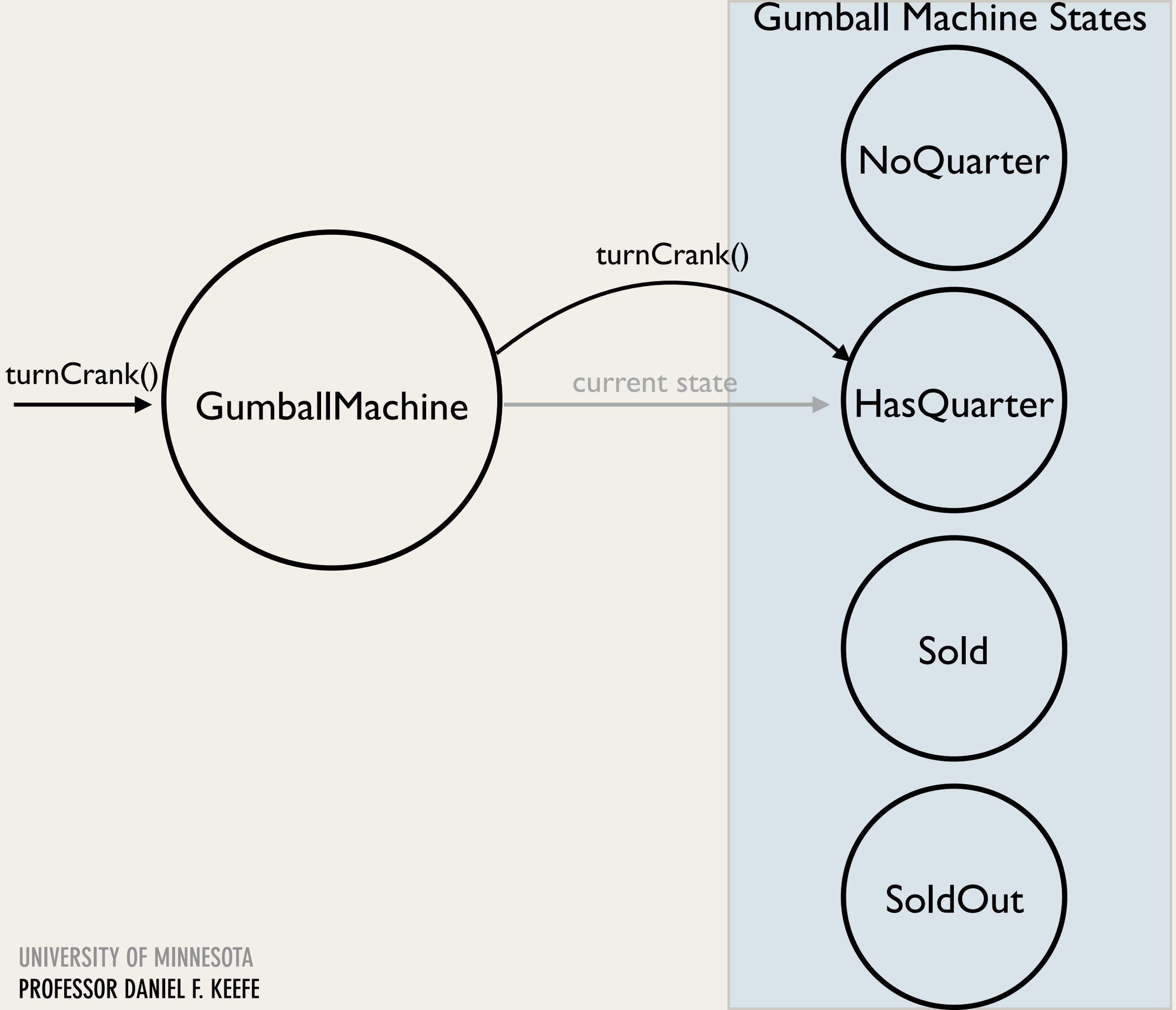
Create a Facade



Why MVC?

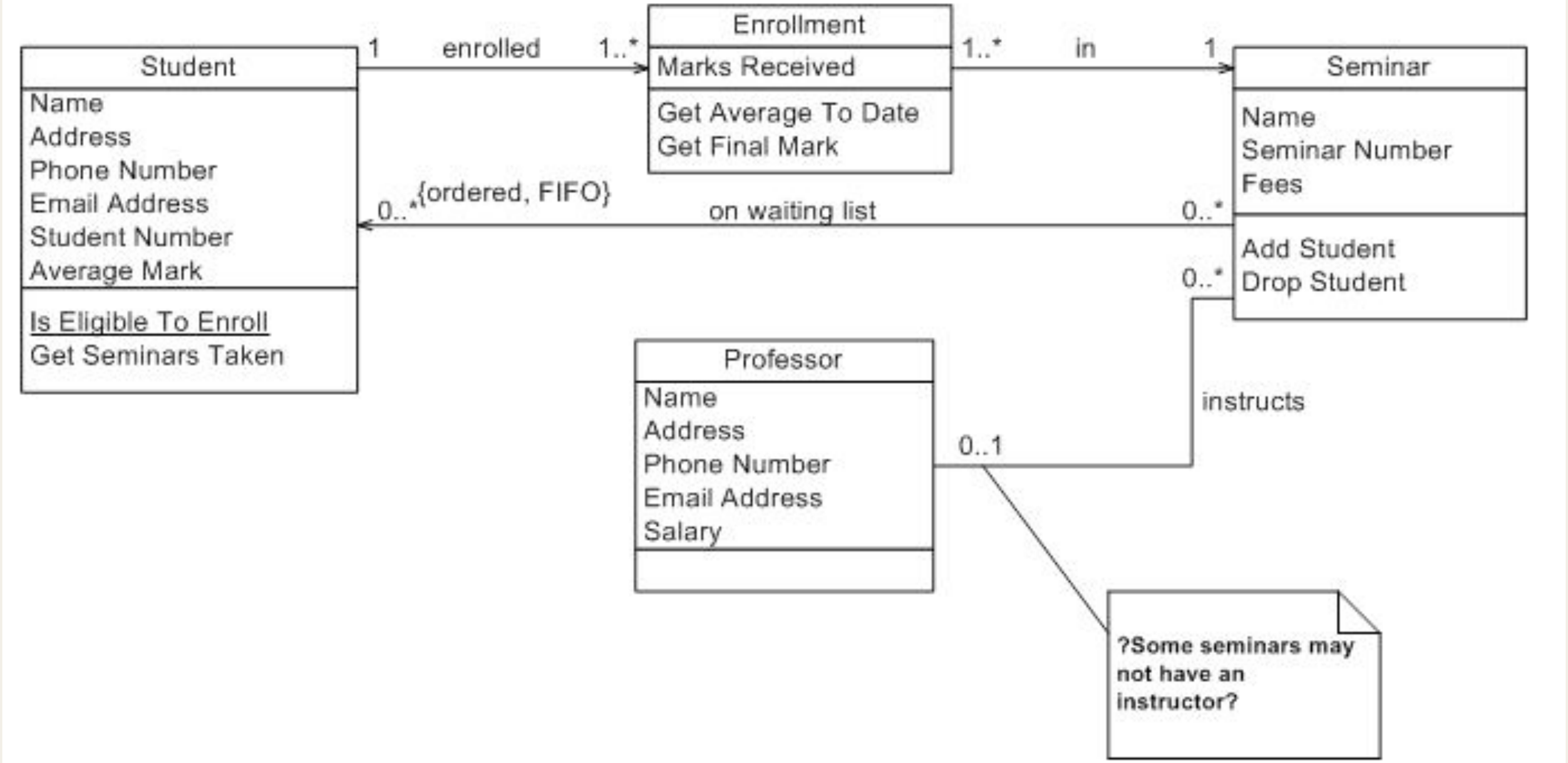
- Mixing all pieces in one place does not scale
- Separation eases maintenance and extensibility
  - Easy to add a new view later
  - Model can be extended, but old views still work
  - Views can be changed later (e.g. add 3D)

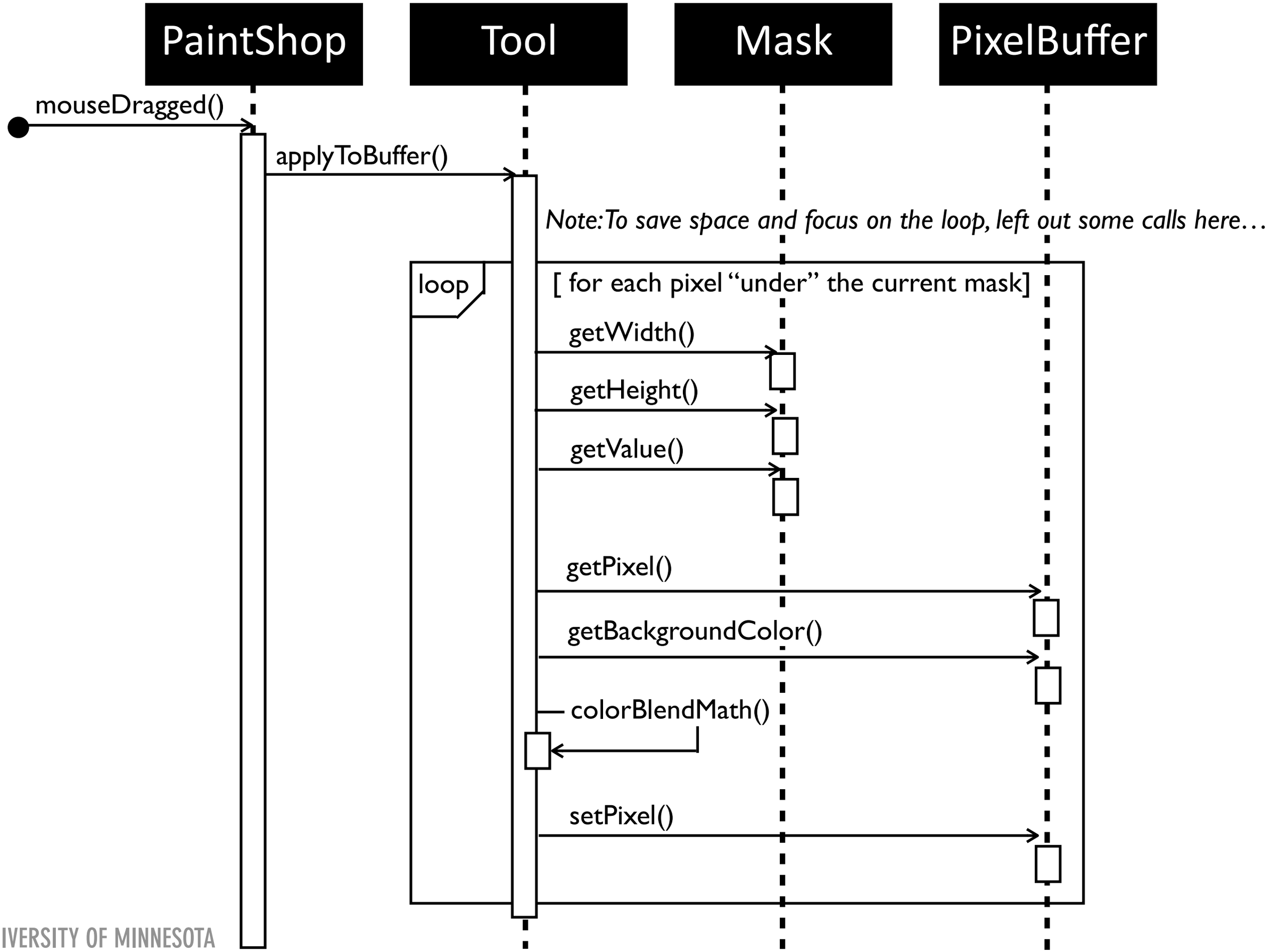



Gumball Machine States
NoQuarter
turnCrank()
turnCrank()
GumballMachine
current state
HasQuarter
Sold
SoldOut

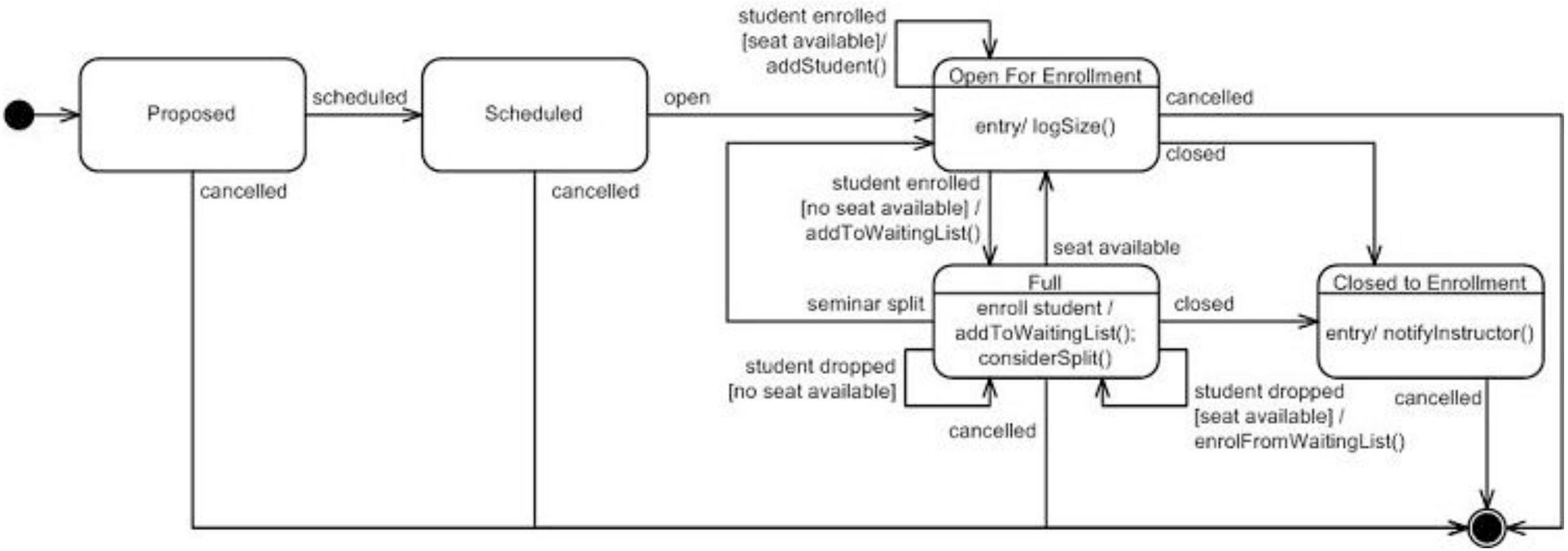
Big Topic #2: Writing (including Diagramming) for Software Construction

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE


Enrollment
Marks Received
Get Average To Date
Get Final Mark
Student
Name
Address
Phone Number
Email Address
Student Number
Average Mark
Is Eligible To Enroll
Get Seminars Taken
1
enrolled
1..*
1..*
in
1
Seminar
Name
Seminar Number
Fees
Add Student
Drop Student
0..*{ordered, FIFO}
on waiting list
0..*
0..*
instructs
Professor
Name
Address
Phone Number
Email Address
Salary
0..1
?Some seminars may not have an instructor?


PaintShop
Tool
Mask
PixelBuffer
mouseDragged()
applyToBuffer()
Note:To save space and focus on the loop, left out some calls here…
loop
[for each pixel "under" the current mask]
getWidth()
getHeight()
getValue()
getPixel()
getBackgroundColor()
colorBlendMath()
setPixel()

State Machine Diagrams



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

Documentation without Comments

```
for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
   isPrime[ primeCandidate ] = true;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
   int factorableNumber = factor + factor;
   while ( factorableNumber <= num ) {
      isPrime[ factorableNumber ] = false;
      factorableNumber = factorableNumber + factor;
   }
}

for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
   if ( isPrime[ primeCandidate ] ) {
      System.out.println( primeCandidate + " is prime." );
   }
}
```

Kinds of Comments

- Repeat of the code
  - basically useless
- Explanation of the code
  - usually explain complicated, tricky, or sensitive pieces of code. Sometimes useful, but if the code is so complicated that it needs to be explained, then might be better to improve the code.
- Marker in the code
  - return NULL; // ***** NOT DONE! FIX BEFORE RELEASE!!!
- Summary of the code
  - summaries a few lines of code.. a lot better than repeating the code
- ****Description of the code's intent
  - // get current employee information (intent) vs.
  - // update employeeRecord object (summarizes the solution)
- Info that is impossible to put into code
  - copyright, confidentiality, version numbers, notes about design, references, etc.

Relevant forms of writing for software projects

1. Source code (i.e., self-documenting code)
2. Source code comments and doxygen-style documentation
3. A project website that provides a project summary
4. Tutorials for users
5. Tutorials for developers
6. Code organization / design guide for developers
7. Coding style guide for developers
8. Developers' blogs

Big Topic #3: C++ Development (including use of source control, makefiles, and external libraries)

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

What you need to learn about building C++ programs

- There are two steps to building a C++ program:
  1. compiling
  2. linking
- Compiling: Every .cpp file that is part of the program needs to be compiled separately into an object file.
  - File1.cpp gets compiled into File1.o
  - File2.cpp gets compiled into File2.o
  - …
- Linking: All of the resulting .o files are then linked together to create an executable program.
  - File1.o, File2.o, and … all get linked together to create MyProgram.exe (or just MyProgram on linux systems)

Class Implementation

Specify the interface in the Date.h file:

```
class Date {
public:
  Date(int y, int m, int d);
  virtual ~Date();
  string print();

private:
  int year;
  int month;
  int day;
};
```

Define the details of what happens when each method is called within the Date.cpp file:

```
Date::Date(int y, int m, int d) {
  year = y;
  month = m;
  day = d;
}

Date::~Date() {
}

string Date::print() {
  cout << year << " " << month
       << " " << day << endl;
}
```

Last time, I asked you to review some example code that uses pointers to C++ classes… why important? questions?

```
// EXAMPLE 6:
// This strategy of working with pointers to a base class is especially
// useful when storing a long list of Ducks.  We'll use the C++ std::vector
// class to create a list of pointers to Ducks.

std::vector<Duck*> duckList;

Duck *duck1 = NULL;
duck1 = new MallardDuck();
duck1->setName("Mallard Junior");
duckList.push_back(duck1);
```

```
// If we want, we can skip the first step of setting the pointer to NULL
// and write the code more compactly like this:
Duck *duck2 = new RubberDuck();
duck2->setName("Squeaky");
duckList.push_back(duck2);

Duck *duck3 = new DecoyDuck();
duck3->setName("Mr. Quiet");
duckList.push_back(duck3);

// Now, we can work with the whole list of ducks very easily
for (int i=0; i<duckList.size(); i++) {
    duckList[i]->fly();
    duckList[i]->performQuack();
}
```

Parameter Passing Example

- Will this work?

```
// Exchange the values of v1 and v2
void swap(int v1, int v2) {
   int tmp = v1;
   v1 = v2;
   v2 = tmp;
}
```

- By default C++ uses **pass-by-value**, which means the called routine cannot modify the original (“outside”) value of v1 and v2.

```
int size = 100;
int *ps = 0;
int size2 = 0;
int *ps2 = 0;
ps = &size;
ps2 = &size2;
size2 = *ps;
if (size == size2)
  cout << "size == size2" << endl;
else
  cout << "size != size2" << endl;
if (ps == ps2)
  cout << "ps == ps2" << endl;
else
  cout << "ps != ps2" << endl;
if (*ps == *ps2)
  cout << "*ps == *ps2" << endl;
else
  cout << "*ps != *ps2" << endl;
```

What does this print out?

Copy Constructors: The Problem in One Sentence

- The compiler's assumption is that copying an object can be done using a bitcopy, and in many cases this may work fine, but in HowMany it doesn't fly because the meaning of initialization goes beyond simply copying the bits.

Const Type Checking: A Key Concept for the Remaining Uses of Const

- You can assign the address of a non-const object to a const pointer.
- But, you **cannot** assign the address of a const object to a non-const pointer -- this breaks the promise that it cannot change.

```
int d = 1;
const int e = 2;
int* u = &d;        // ok because d is not const
int* v = &e;        // Illegal, e is const, but v is not
```

- Define a new template function called “Add(n1, n2)”
- Add two objects together, store the result in a local variable within your function.
- Then, return that local variable.
- For reference:

```
template<typename TYPE>
TYPE Twice(TYPE data) {
    return data * 2;
}

cout << Twice(10);
cout << Twice(3.14);
```

Example with Iterators

```
vector<Page> book;

// Add a bunch of pages to the book;
...
```

```
vector<Page>::iterator itr;
for (itr = book.begin(); itr != book.end(); itr++) {
  // Assume that operator<<() is defined for Page
  cout << *itr << " ";
}
cout << endl;
```

```
void assert (int expression);
```

If the expression is false, a message is written to the standard error device and abort is called, terminating the program execution.

```
#include <stdio.h>      // printf
#include <assert.h>     // assert

void print_number(int* myInt) {
  assert(myInt != NULL);
  printf("%d\n",*myInt);
}
```

The specifics of the message at least include: the expression whose assertion failed, the name of the source file, and the line number where it happened:

```
Assertion failed: expression, file filename, line line number
```

The macro is automatically disabled when NDEBUG is defined, so when you compile your code in "release mode", you should define this flag so all the debugging assertions will not slow down your program, e.g.:

```
g++ -c -D NDEBUG myprogram.cpp
```

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/

Makefiles: Use Case

Suppose you want to compile helloworld.
You typed **`make all`** into your shell.
What happens?

Makefile

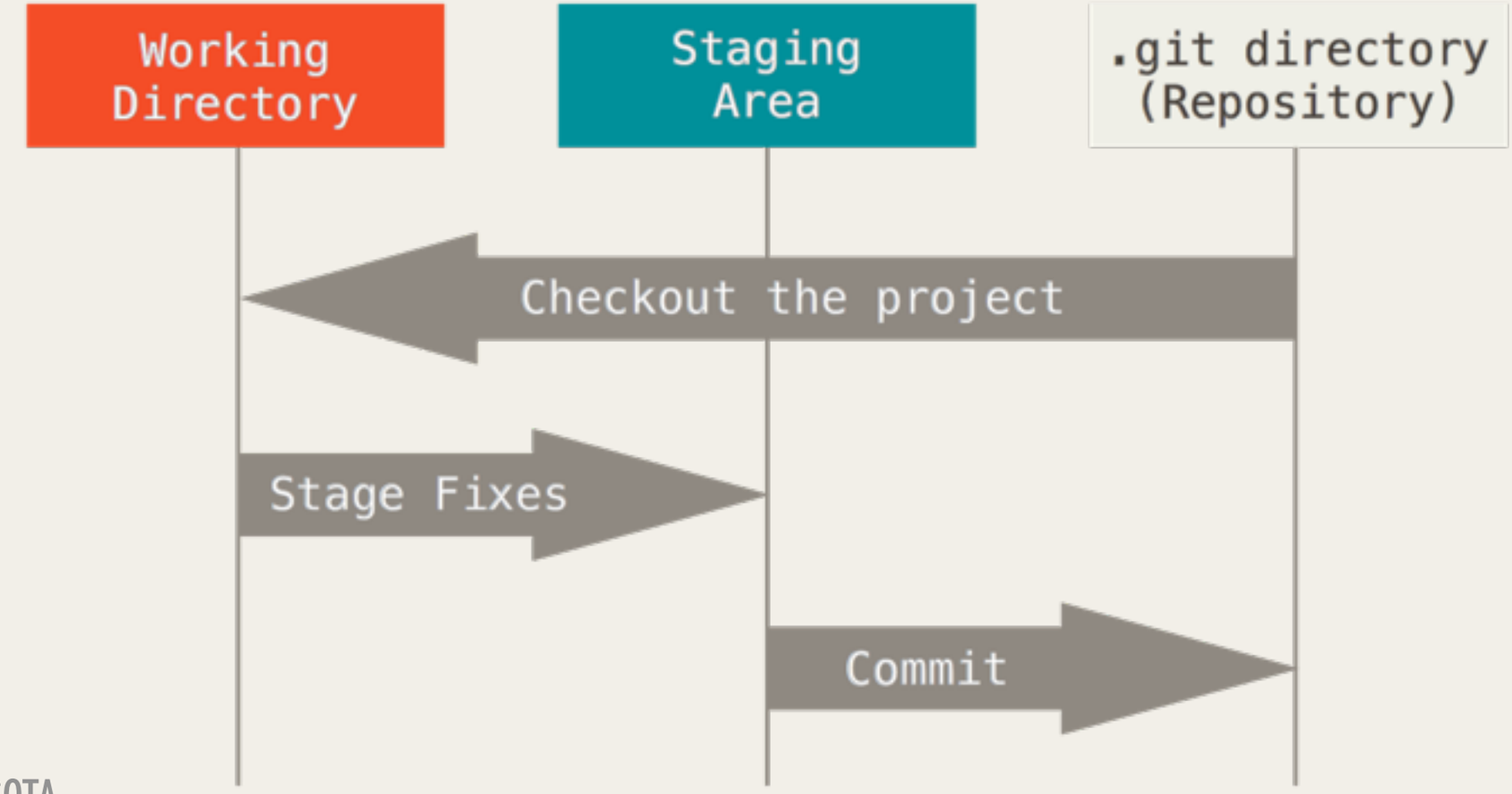
```
all: helloworld # default target

helloworld: main.o # linking rule
    g++ -o helloworld main.o

main.o: main.cpp # compile rule
    g++ -c main.cpp

clean:
    rm helloworld main.o
```

Makefiles: Let's talk about dependencies.

- Does "hello.txt" depend upon anything?
- Add "name.txt" as a dependency for the "hello.txt" rule in your Makefile.
- Now, try running "make hello.txt" a few times. What happens? What happens if you edit name.txt then re-run make?

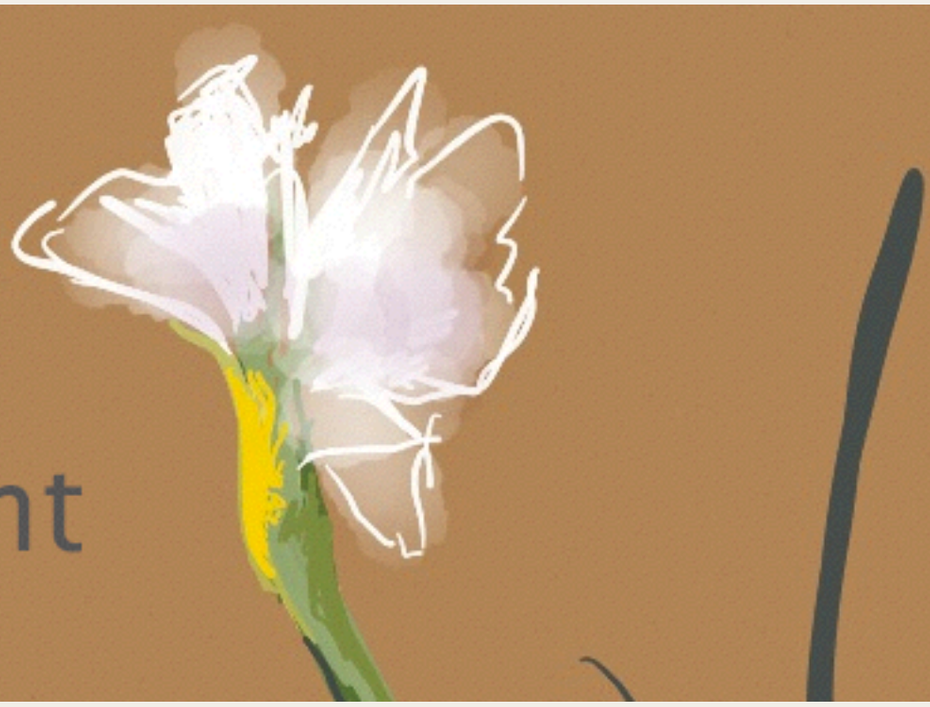
Source Control (git): Typical Workflow

1. You modify files in your working directory.
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores the snapshot permanently in your .git directory.



CSci-3081

Program Design and Development



UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

the end