COMP 3522

Object Oriented Programming in C++

Week 9 Day 2

Agenda

1. Genetic Algorithm
   - Assignment 2
2. Design patterns intro
   1. Singleton
   2. Observer

COMP 3522

GENETIC ALGORITHMS

Genetic algorithms

- Computing has been helpful in many problem domains
- Many problem domains have, in turn, lent problem solving strategies to computing
- Genetic algorithms are inspired by the process of natural selection described in contemporary biology
- Useful for large problems with solutions that are difficult to find
- Useful for optimization, search problems.

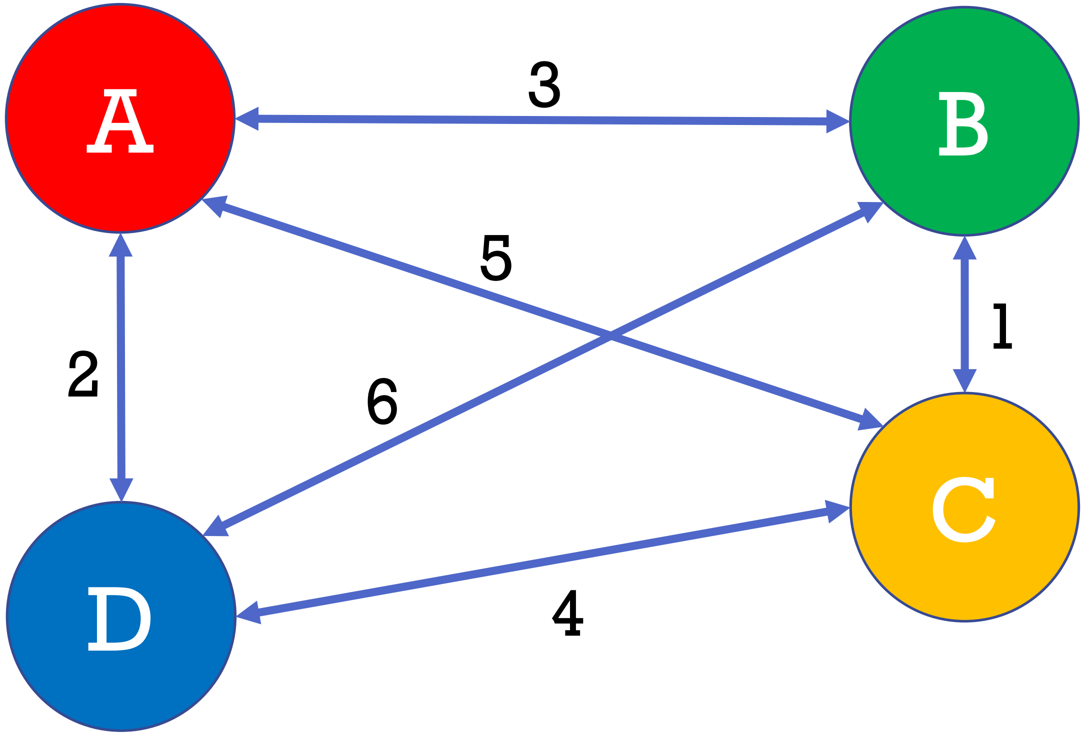
Genetic algorithm

```
{
        initialize population
        evaluate population
        while (termination criteria not reached)
        {
                select solutions for next population
                perform crossover and mutation
                evaluate population
        }
}
```

GENETIC ALGORITHM: Travelling salesperson

Travelling salesperson problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?



Travelling salesperson problem

- Input:
  - List of cities to visit
- Requirements:
  - Visit all the cities
    - Return to original city
  - Minimize travelling distance

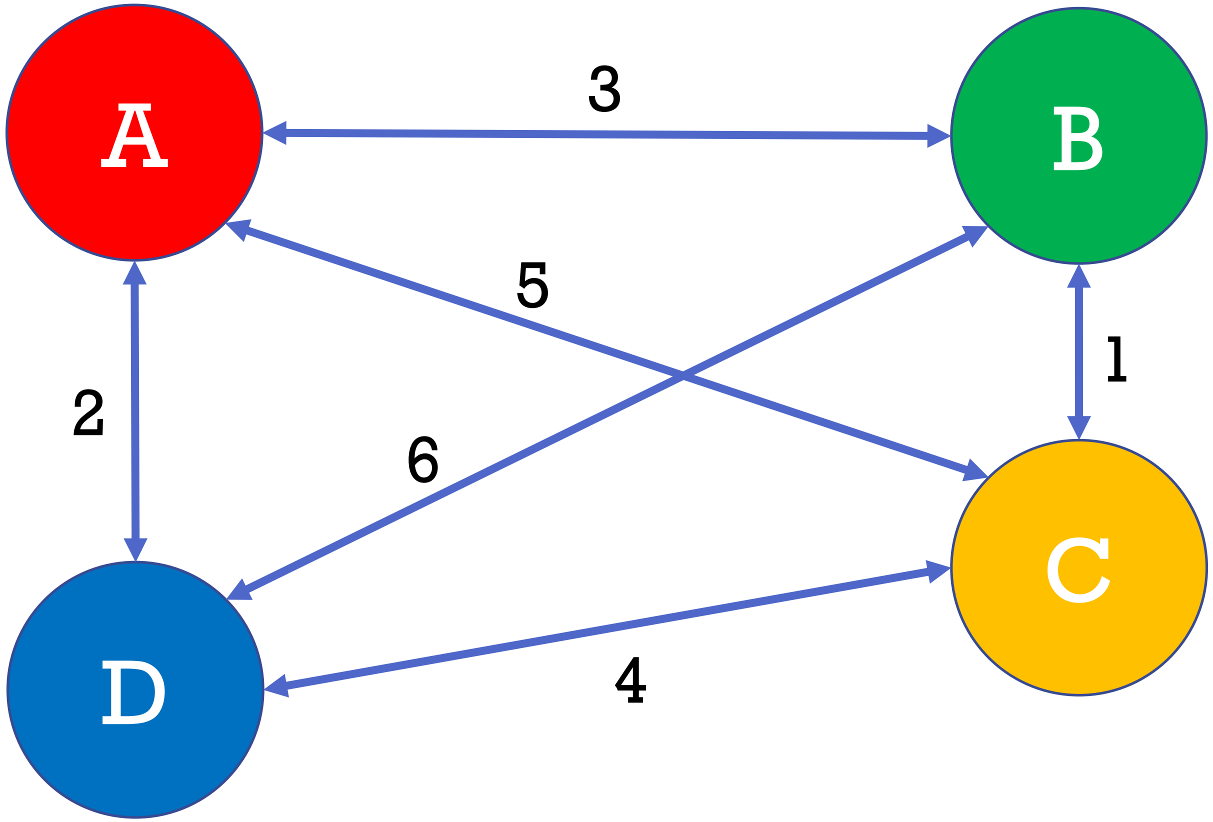
This sounds easy (we can do this by hand with a few cities)

O(**n!**) – this becomes impractical with 20 cities

What if we have 200 cities, or 200,000 cities, or ALL the cities and towns and villages in the world?

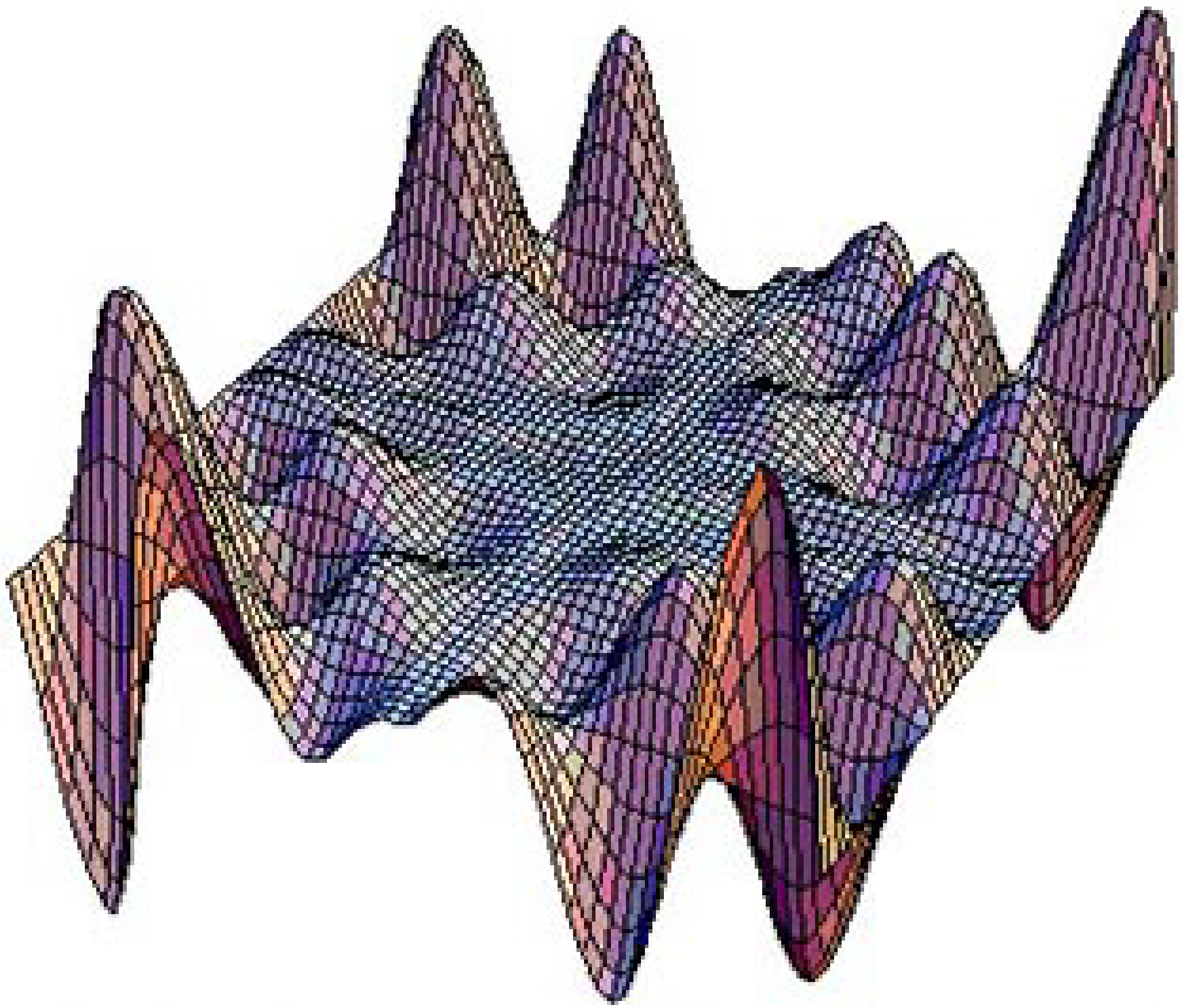
Let's travel around 4 cities

- **ABCDA = 3+1+4+2 = 10**
- ACBDA = 5+1+6+2 = 14
- ADBCA = 2+6+1+5 = 14
- **ADCBA = 2+4+1+3 = 10**
- ...
- DBCAD = 6+1+5+2=14
- DBACD = 6+3+5+4=18



- 24 possibilities = 4! (4 factorial: 4 x 3 x 2 x 1)

Imagine the ‘solutionscape’



http://classes.yale.edu/fractals/CA/GA/Fitness/FitnessLandscape.gif

Some tours are very short

Some tours are very long

There are too many to find a global solution.

OUR GENETIC ALGORITHM: Terminology

Some terms we are using

City: A location that has a name and x/y coordinates.

Cities_to_visit: an invariant (unchanging) list of City structs that we want to visit. The "master list"

Tour: a list of pointers to the cities we want to visit. We can shuffle the pointers easily to compare different orderings of cities without modifying the "master list"

Population: a collection of candidate Tours. We keep the population "sorted," i.e., the "fittest" tours are at the front of the list.

Some terms we are using

Fitness: Each candidate Tour in the population has a fitness, i.e., how "good" it is. For us, a fit Tour has a short travel distance. A Tour with a shorter distance has better fitness.

Elite: Each generation, we can designate one or more Tours that are so amazing they don't cross, they get carried over to the next 'generation'. These Tours are "elite."

Parents: Each iteration, we select some parents from the Population of Tours and use the parents' contents to generate a new Tour for the next iteration.

Some terms we are using

Crosses and Crossover: Each generation we create new Tours by crossing "parents." The crossover algorithm is basic.

Mutation: Each iteration, we randomly "mix up" a few of the Tours in our population. This mimics the random mutations that take place as cells divide, etc.

Mutation rate: If we 'roll' less than the rate, we swap a few cities in the Tour being mutated.

OUR GENETIC ALGORITHM

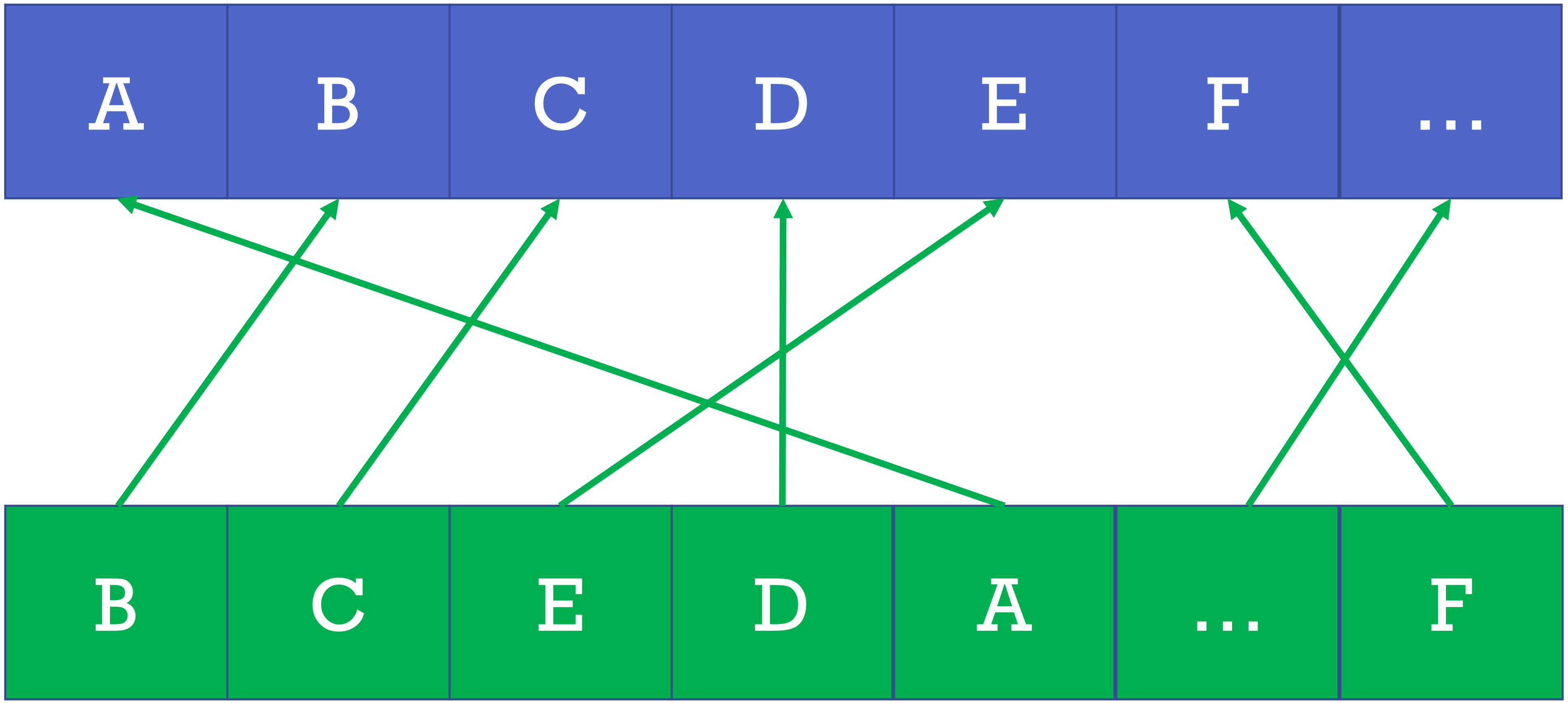
Our genetic algorithm

```
{
    create cities and tours
    evaluate tours' fitness
    while ((fitness < improvement) and (iterations < 1000))
    {
        move elite to front
        perform crossover and mutation of tours
        evaluate tours' fitness
    }
}
```

Our algorithm

1. Create our master list of cities named A, B, C, …, R, S, T. Our master list is 20 Cities long.
2. Create a Population of Tours. The Population contains 30 candidate Tours. Each Tour contains pointers that point to the cities in the master list. Each Tour is shuffled randomly.

Master city list – doesn’t change



Tour – pointers to cities

Have 30 tours each with random pointers to cities

Our algorithm

3. Find the shortest travelling distance in the randomly shuffled Tours. That's our starting point.

Population – list of tours

Tour 1 = 100 cost	
Tour 2 = 200 cost	
…	
Tour 29 = 70 cost	ELITE!
Tour 30 = 150 cost	

Our algorithm

4. (Loop) While we haven't reached our goal (or still have iterations)
   1. Find the best tour in the Population, call it an Elite, and move it to the front of the list so we can keep an eye on it.

Population – list of tours

Population – list of tours	
Tour 29 = 70 cost	Move ELITE to front
Tour 2 = 200 cost	
…	
Tour 28 = 100 cost	
Tour 30 = 150 cost	

- Identified the best tour of the existing tours
- Next create a new list of tours based on the existing tours

OUR GENETIC ALGORITHM: Crossing

Our algorithm

4. (Loop) While we haven't reached our goal (or still have iterations)
    2. Create a temporary list of Tours called Crosses.
    3. Generate a new Tour by crossing parents for each remaining Tour in the original population

Original Population

Elite Tour = 70 cost
Tour 2 = 200 cost
…
Tour 28 = 100 cost
Tour 30 = 150 cost

Crosses – new population

Elite Tour = 70 cost
?
?
?
?

Step 4.3 Crossing parents

Original Population

Elite Tour = 70 cost
Tour 2 = 200 cost
…
Tour 28 = 100 cost
Tour 30 = 150 cost

Set 1

Tour 26 = 500 cost
Tour 11 = 700 cost
Tour 27 = 800 cost
Tour 14 = 900 cost
Tour 30 = 150 cost

Set 2

Tour 13 = 700 cost
Tour 2 = 200 cost
Tour 12 = 300 cost
Tour 10 = 600 cost
Tour 9 = 350 cost

- Pick **two sets** of **5 random tours** from the original population

Step 4.3 Crossing parents

- Find the fittest tour in each set.
- These **two parents** will be crossed to generate a new child

Set 1	
Tour 26 = 500 cost	
Tour 11 = 700 cost	
Tour 27 = 800 cost	
Tour 14 = 900 cost	
Tour 30 = 150 cost	**Parent 1**

Set 2	
Tour 13 = 700 cost	
Tour 2 = 200 cost	**Parent 2**
Tour 12 = 300 cost	
Tour 10 = 600 cost	
Tour 9 = 350 cost	

Step 4.3 Crossing parents

- WARNING – All tours should have **20 cities** in this example:
- ie:
  - Tour 30 [A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T]
  - Tour 2 [D,C,A,B,E,S,G,H,I,T,K,L,N,M,O,P,F,R,Q,J]
- But we're shortening it to **5 cities** a tour for demonstration purposes

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Child tour
?????

Step 4.3 Crossing parents

- Pick a random index and copy all cities up to and including that index from parent 1
  - Randomly pick index 1. Start from beginning of **Parent 1**, copy everything up to and including index 1 from **Parent 1** to Child

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Child tour
?????

Step 4.3 Crossing parents

- Pick a random index and copy all cities up to and including that index from parent 1
  - Randomly pick index 1. Start from beginning of **Parent 1**, copy everything up to and including index 1 from **Parent 1** to Child

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Child tour
A????

Step 4.3 Crossing parents

- Pick a random index and copy all cities up to and including that index from parent 1
  - Randomly pick index 1. Start from beginning of **Parent 1**, copy everything up to and including index 1 from **Parent 1** to Child

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Child Tour
AB???

Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities in **Parent 2** and **Child**. Copy over non-duplicate cities

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Child Tour
ABD??

Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities in **Parent 2** and **Child**. Copy over non-duplicate cities

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Child Tour
ABDC?

Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities in **Parent 2** and **Child**. Copy over non-duplicate cities

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Skip duplicate A

Child Tour
ABDC?

Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities in **Parent 2** and **Child**. Copy over non-duplicate cities

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Skip duplicate B

Child Tour
ABDC?

Step 4.3 Crossing parents

- After hitting index 1 of parent 1, start from beginning of parent 2
  - Skip duplicate cities in **Parent 2** and **Child**. Copy over non-duplicate cities

Parent 1

Tour 30 = 150 cost
A B C D E

Parent 2

Tour 2 = 200 cost
DCABE

Child Tour
ABDCE

Our algorithm – insert merged tours

- This is our **new merged tour**. Repeat previous steps for the rest of the new population of tours

Merged Tour 1
ABDCE

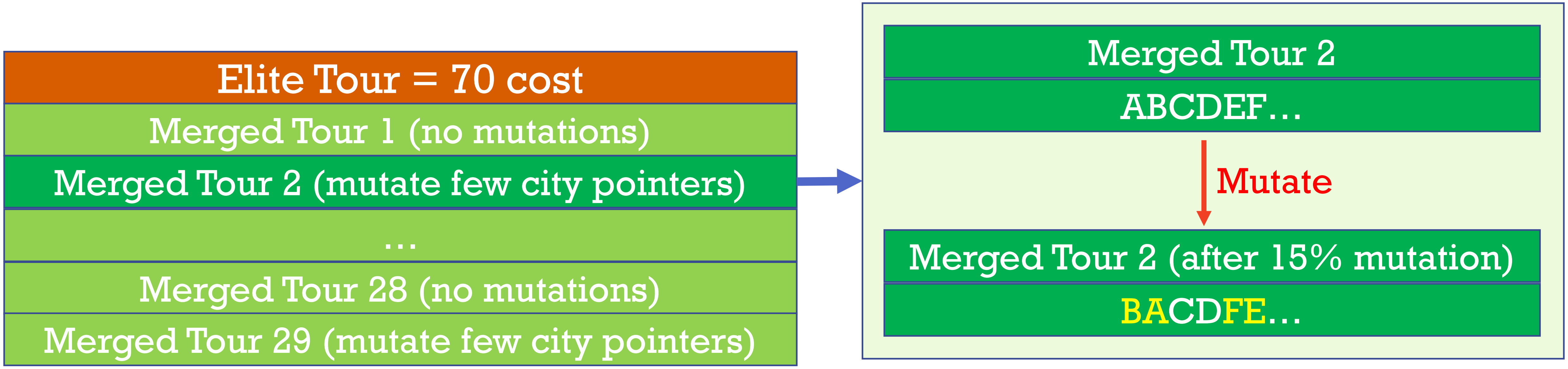
4. (Loop) While we haven't reached our goal (or still have iterations)
    4. Replace all the Tours in our Population (except the Elite Tour) with the new crosses

Elite Tour = 70 cost
Merged Tour 1
Merged Tour 2
…
Merged Tour 28
Merged Tour 29

OUR GENETIC ALGORITHM: Mutating

Our algorithm – mutate population

4. (Loop) While we haven't reached our goal (or still have iterations)
   5. Mutate some of the population (except the Elite) by swapping around some of the Cities in each Tour.



Our algorithm – evaluate fitness

4. (Loop) While we haven't reached our goal (or still have iterations)
   6. Evaluate the fitness (distance) and report it.

Elite Tour = 70 cost
Merged Tour 1
Merged Tour 2
Merged Tour …
Merged Tour 28
Merged Tour 29

Elite Tour = 70 cost	
Merged Tour 1 = 300 cost	
Merged Tour 2 = 50 cost	ELITE!
Merged Tour …	
Merged Tour 28 = 170 cost	
Merged Tour 29 = 210 cost	

5. Bam. You just implemented a genetic algorithm in C++!

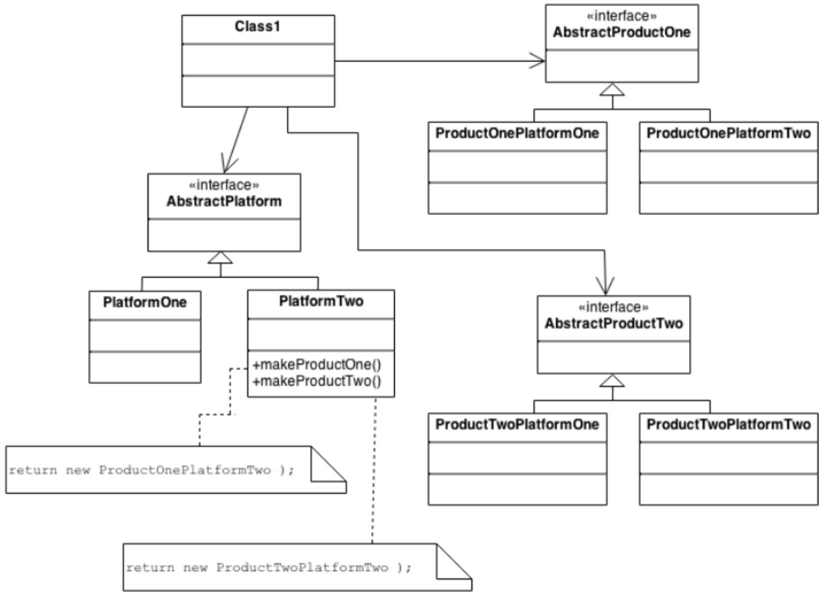
DESIGN PATTERNS

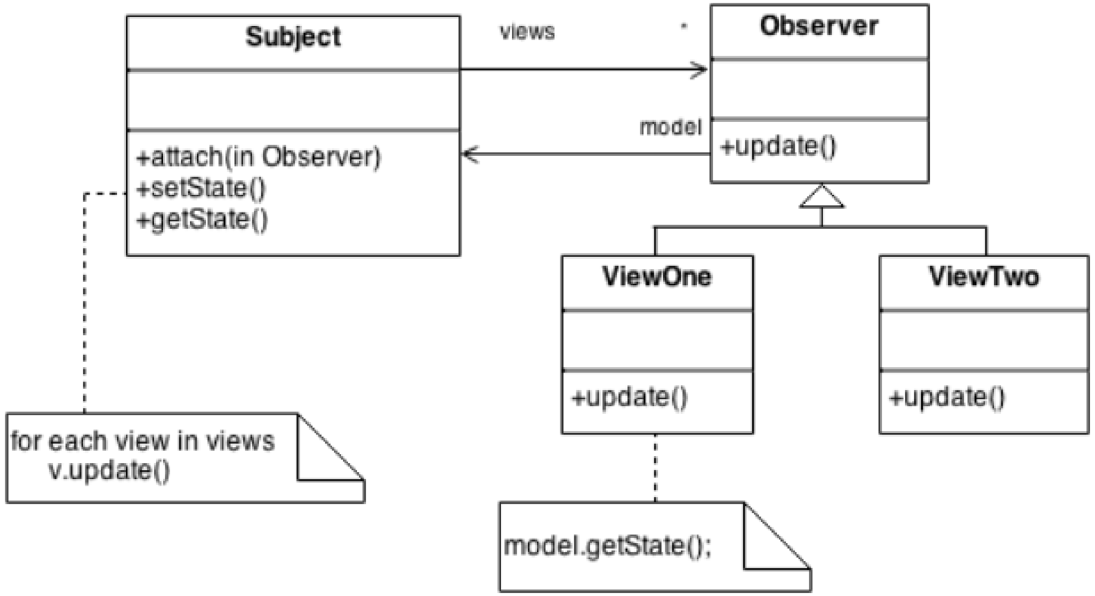
What are Design Patterns

- Common design solutions to common architectural problems
- How can I write systems so:
  - Classes can **communicate** with each other with low coupling?
  - Classes can be combined to form new **structures?**
  - Classes can be **created** with different strategies and techniques?

What are Design Patterns

- Think of these as recipes or templates to solve common design problems
- “If your classes/code is structured in this way, it will solve a specific design issue”





Design Patterns - Advantages

- Don’t re-invent the wheel, use a proven solution instead
- Are abstract, and can be applied to different problems
- Communicate ideas and concepts between developers
- Language agnostic. Can be applied to most (if not all) OOP programs.

Design Patterns - Disadvantages

- Can make the system more complex making the system harder to maintain. Patterns are deceptively 'simple'.
- The system may suffer from pattern overload.
- All patterns have some disadvantages and add constraints to a system. As a result a developer may need to add a constraint they did not plan for.
- Do not lead to direct code re-use.

Categorizing Design Patterns

- **<u>Behavioural</u>**

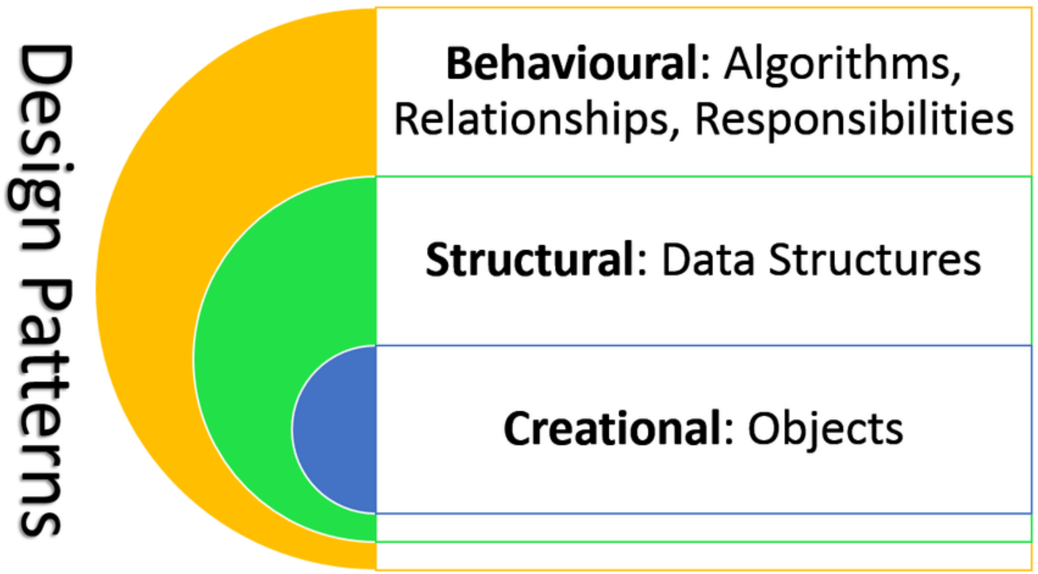
Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

- **<u>Structural</u>**

How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- **<u>Creational</u>**

All about class instantiation. Different strategies and techniques to instantiate an object, or group of objects



Picking a Pattern

Step 1	• Understand the problem you are facing in terms of dependencies, modularity and abstract concepts.
Step 2	• Identify if this is a behavioural, structural or creational issue?
Step 3	• Are there any constraints that I need to follow?
Step 4	• Is there a simpler solution that works? If not, pick a pattern.

SINGLETON

Introduction

- I need a system where there is a single object that is accessible from anywhere in the code

- **How do I do this without global variables?**

Singleton design pattern: a really easy one!

- **Design pattern category: Creational**
- Sometimes we want to guarantee that only a **single instance** of a class will ever exist
- We want to prevent more than one copy from being constructed
- We must write code that enforces this rule
- We want to employ the **Singleton Design Pattern**

Singleton pattern

1. **Instantiates** the object on its first use
2. **Ideally hides** a private initializer
3. **Reveals** a public `get_instance` function that returns a reference to a static instance of the class
4. **Provides** "global" access to a single object

Why/how do we use it?

Use the singleton pattern **when you need to have one and <u>only one</u> object of a type** in a system.

Singleton is a globally accessible class where we guarantee only a single instance is created

<u>That's it.</u>

<u>Really, that's all there is to it.</u>

Code sample (so easy!)

```
class singleton
{
    public:
        static singleton& get_instance()
        {
            static singleton instance;
            return instance; // Instantiated on first use.
        }
    private:
        int test_value;
        singleton() {}

    public:
        singleton(singleton const&)         = delete;
        void operator=(singleton const&)   = delete;
        int get_value() { return test_value++; }
};
```

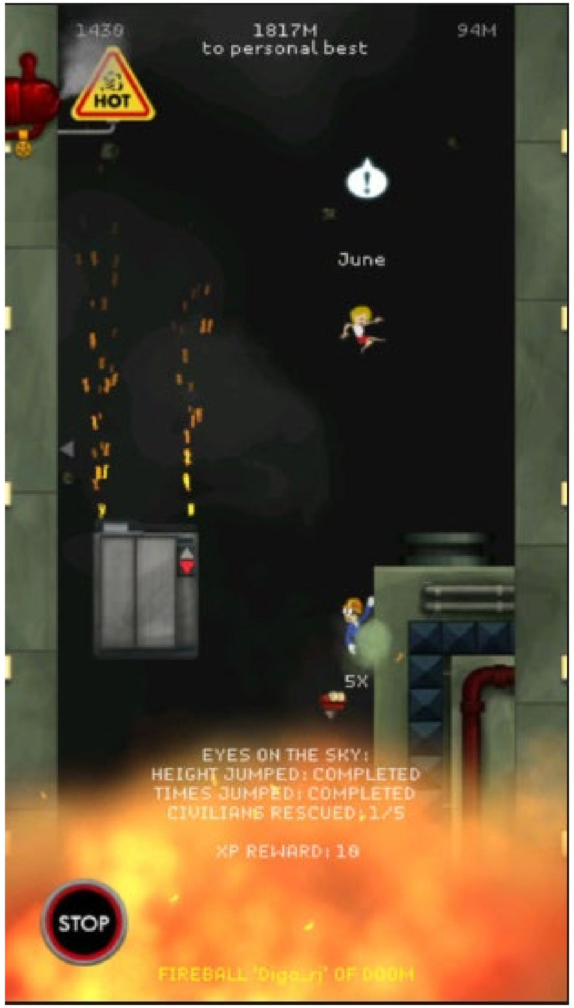
singleton.cpp

Application – Game screen management

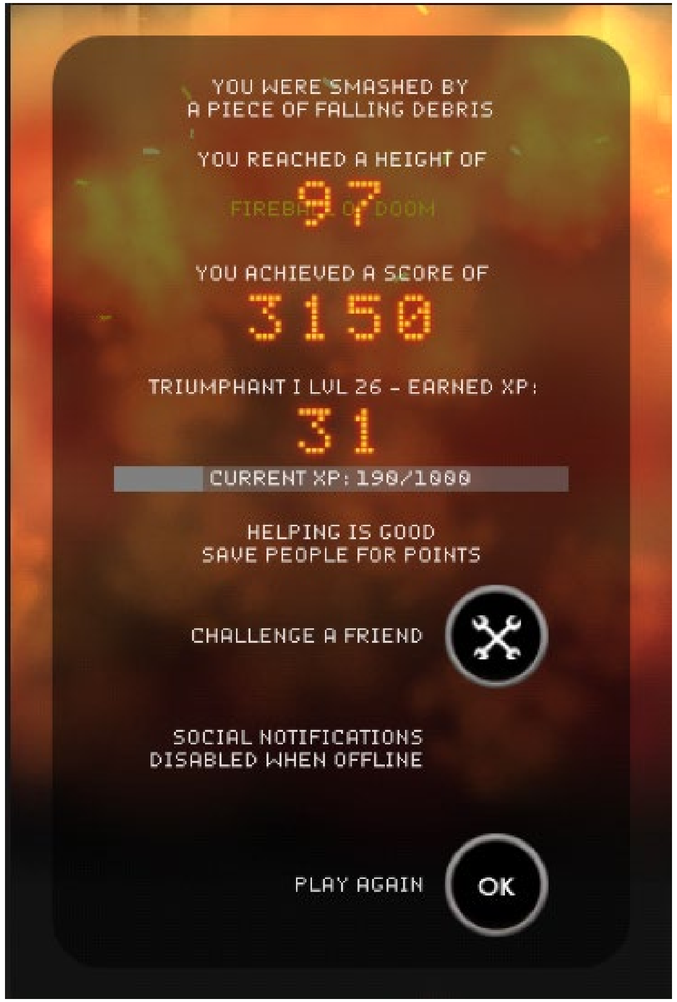
- Game has multiple screens
  - Start, gameplay UI, game over, store, etc
- Different screens must be able to be displayed at various places in the code
  - Store class wants to show store screens
  - Gameplay logic wants to show start/gameplay/game over
  - Settings logic wants to show settings screen
- Need a central place to call and load specific screens on demand

Mechanic Panic – Singleton screens example

GameState enum: MAIN_MENU
ScreenManager::getInstance().show(MAIN_MENU);



GameState enum: GAMEPLAY
ScreenManager::getInstance().show(GAMEPLAY);

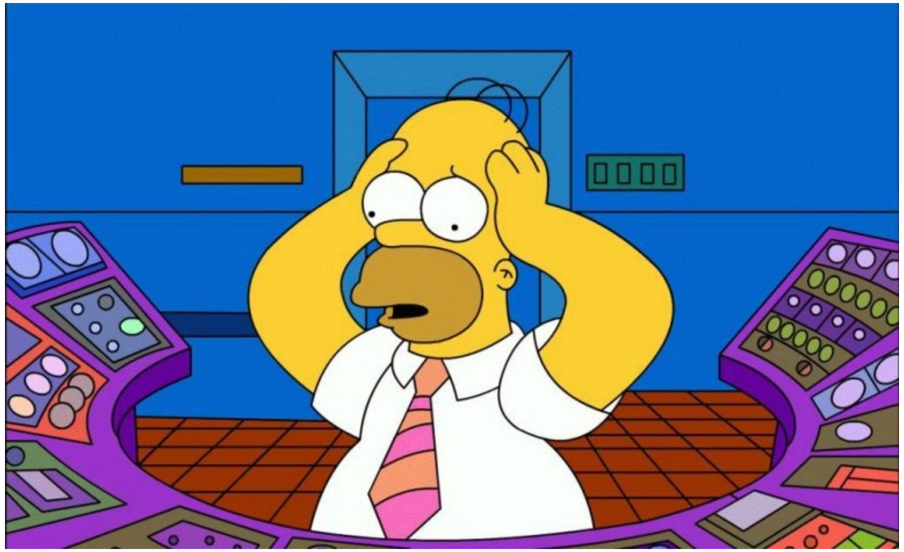


GameState enum: GAME_OVER
ScreenManager::getInstance().show(GAME_OVER);

OBSERVER

Introduction

- I want to create a system where one object can broadcast information to multiple objects
- How do I notify a bunch of different kinds of objects if the state of one part of the system changes without **tightly coupling** that part of the system with the rest?



Introduction

- We like to partition our systems into cooperating classes
- Those classes share information
- We need to maintain consistency
- But we can't couple them tightly because that reduces their flexibility
- We use an idiom you will see often in programming called Publish-Subscribe:
  - The **subject** publishes notifications without knowing who **observes**
  - Any number of **observers** can subscribe to receive notifications*

* Sounds a little like Java GUI listeners to me!

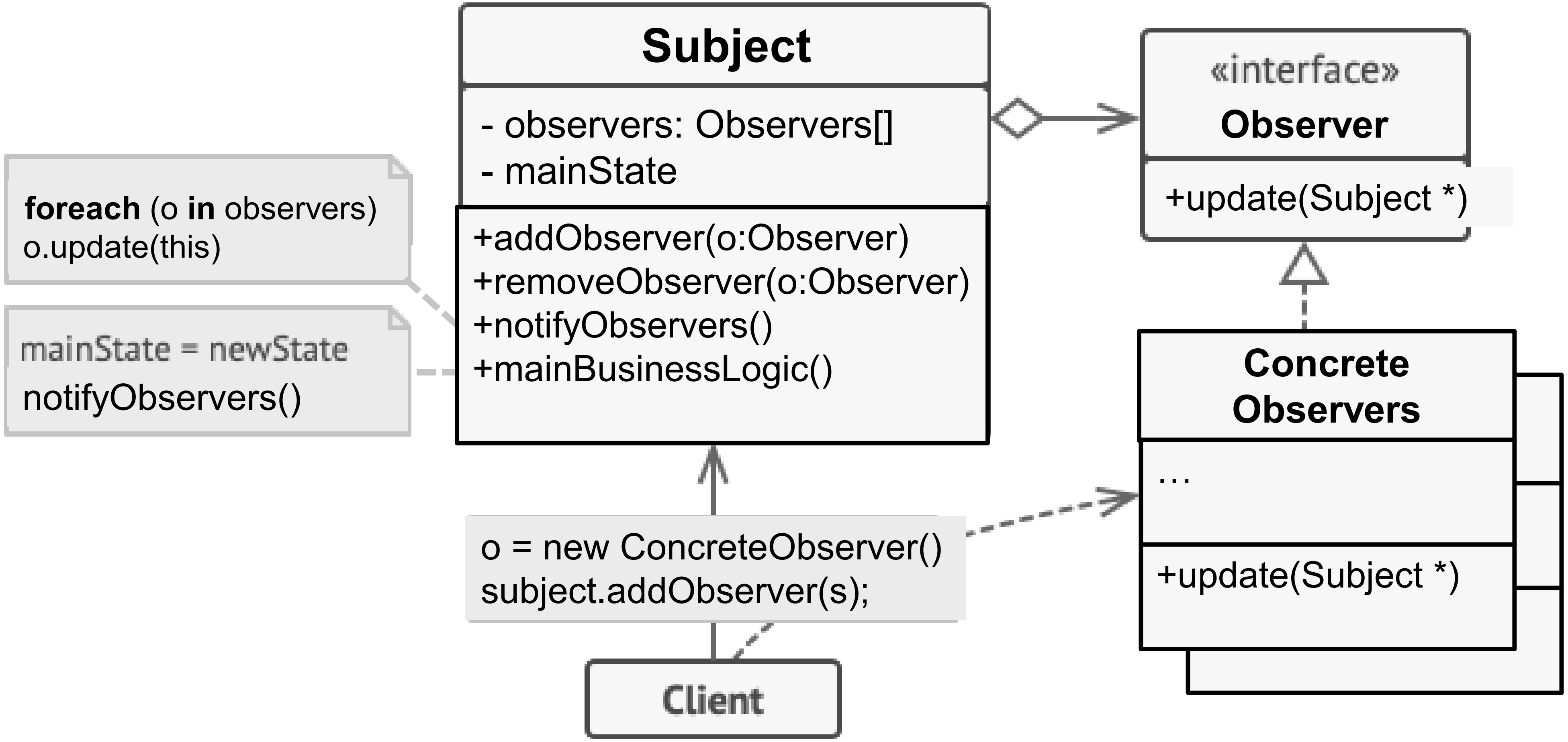
Observer design pattern

- **Design pattern category: Behavioral**
- The **Observer pattern** describes how to establish these relationships
- There are two key objects:
  1. **Subject** may have any number of dependent observers
  2. **Observers** are all notified whenever the subject undergoes a change of state
- Each observer queries the object to synchronize their states
- Observer ensures that when a subject changes state all its dependents are automatically notified

Observer design pattern

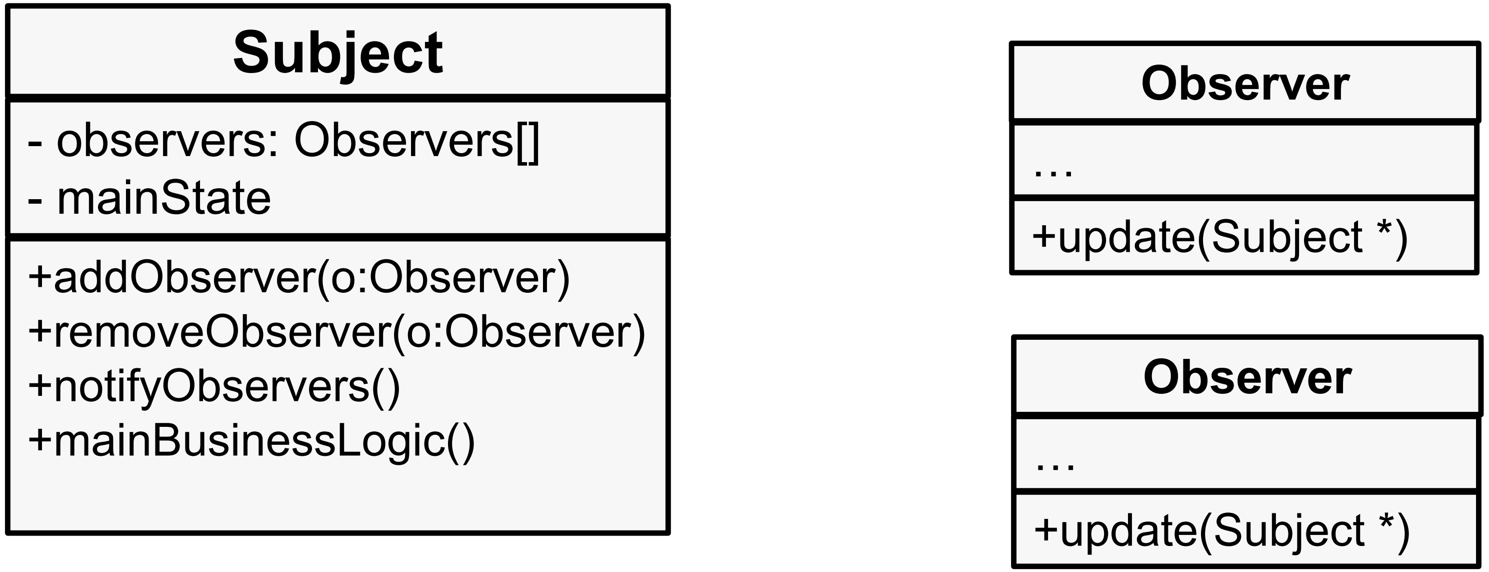
1. (Abstract) Subject
   - Knows its Observers
   - Any number of Observers may observe a subject
   - Provides an interface for attaching and detaching Observers
2. (Abstract) Observer
   - Defines an updating interface for objects that should be notified of changes in a subject
3. ConcreteSubject
   - Sends notification of its changed state to its observers
4. ConcreteObserver
   - Maintains a reference to a ConcreteSubject object
   - Stores state that needs to be consistent with the subject's
   - Implements the Observer updating interface

Observer design pattern: Class diagram



https://refactoring.guru/design-patterns/observer

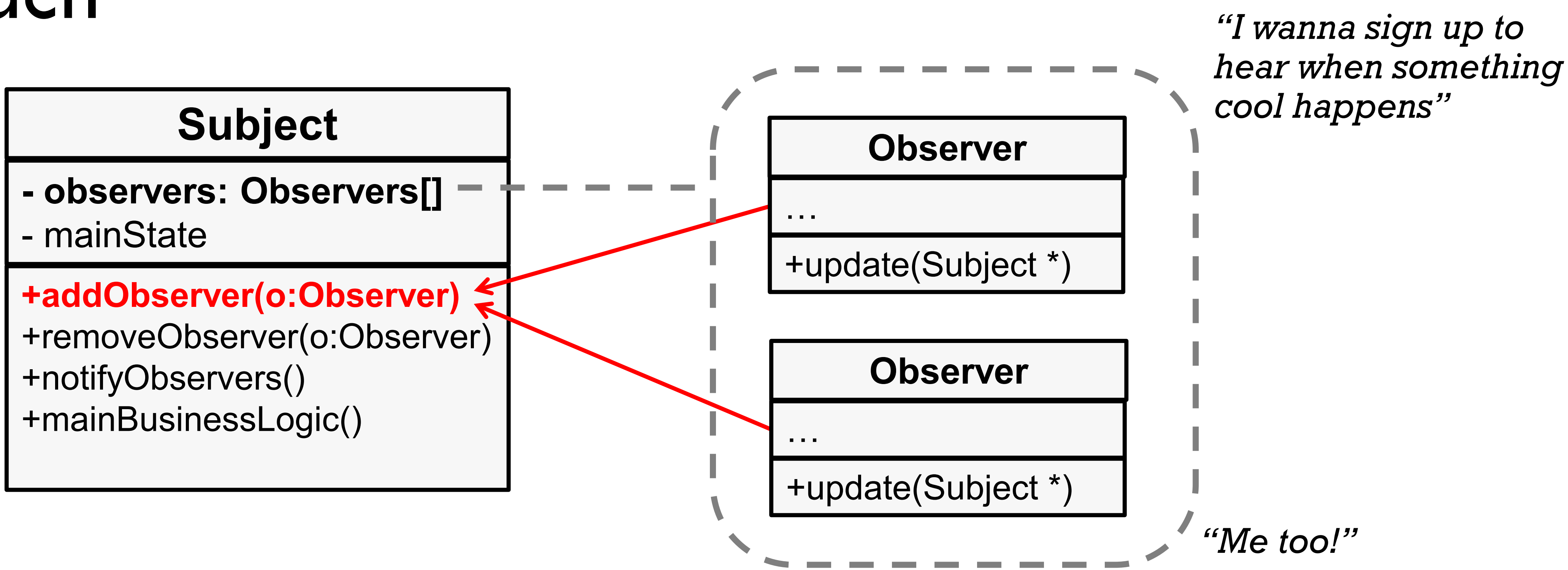
1. Create subject and observers



Instantiate subject, and 2 observer objects

https://refactoring.guru/design-patterns/observer

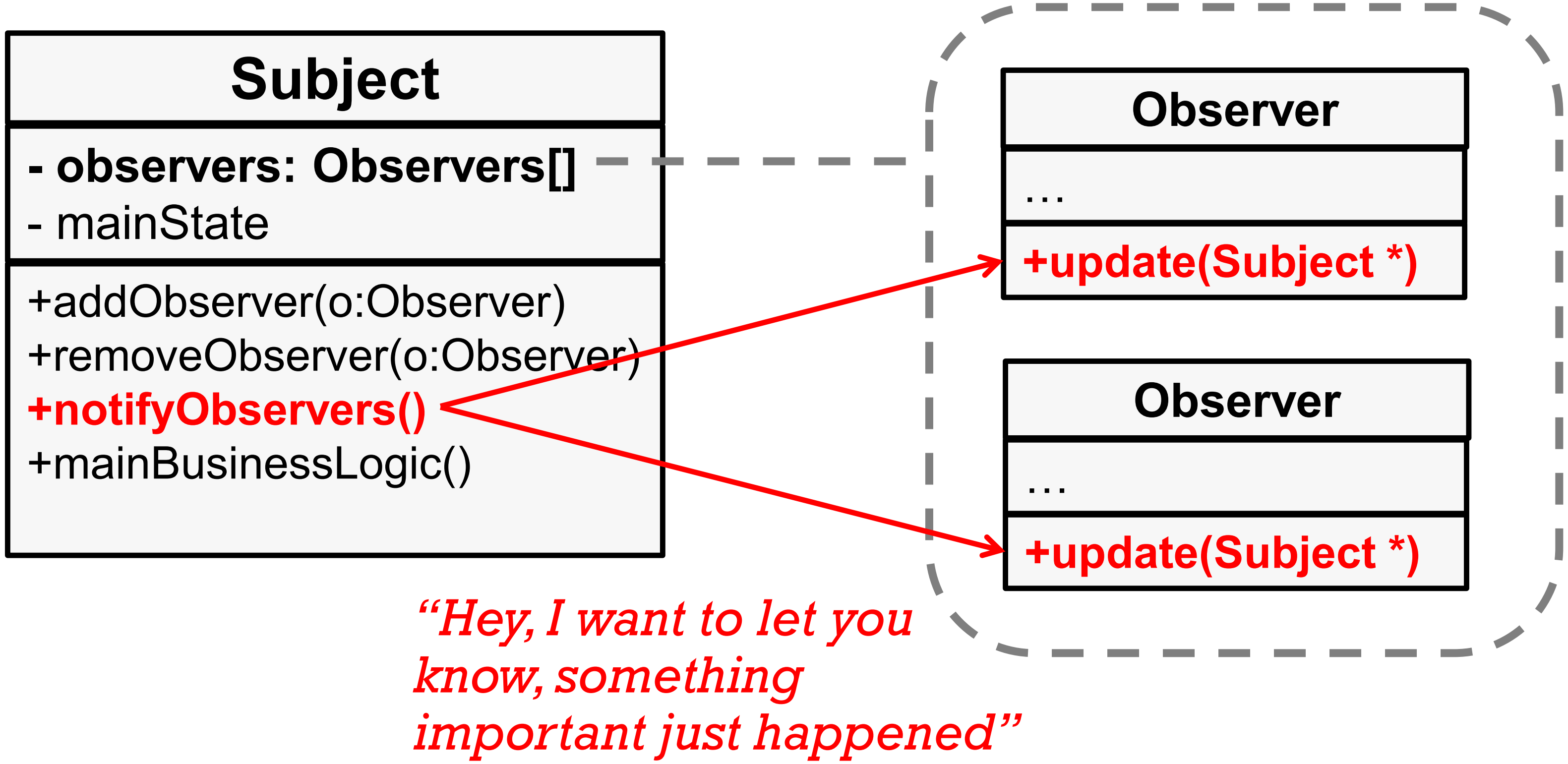
2. Attach



Observer objects can be added to a Subject to "listen in" to important events

https://refactoring.guru/design-patterns/observer

3. Update



Subject notifies objects when important event happens

https://refactoring.guru/design-patterns/observer

Use cases

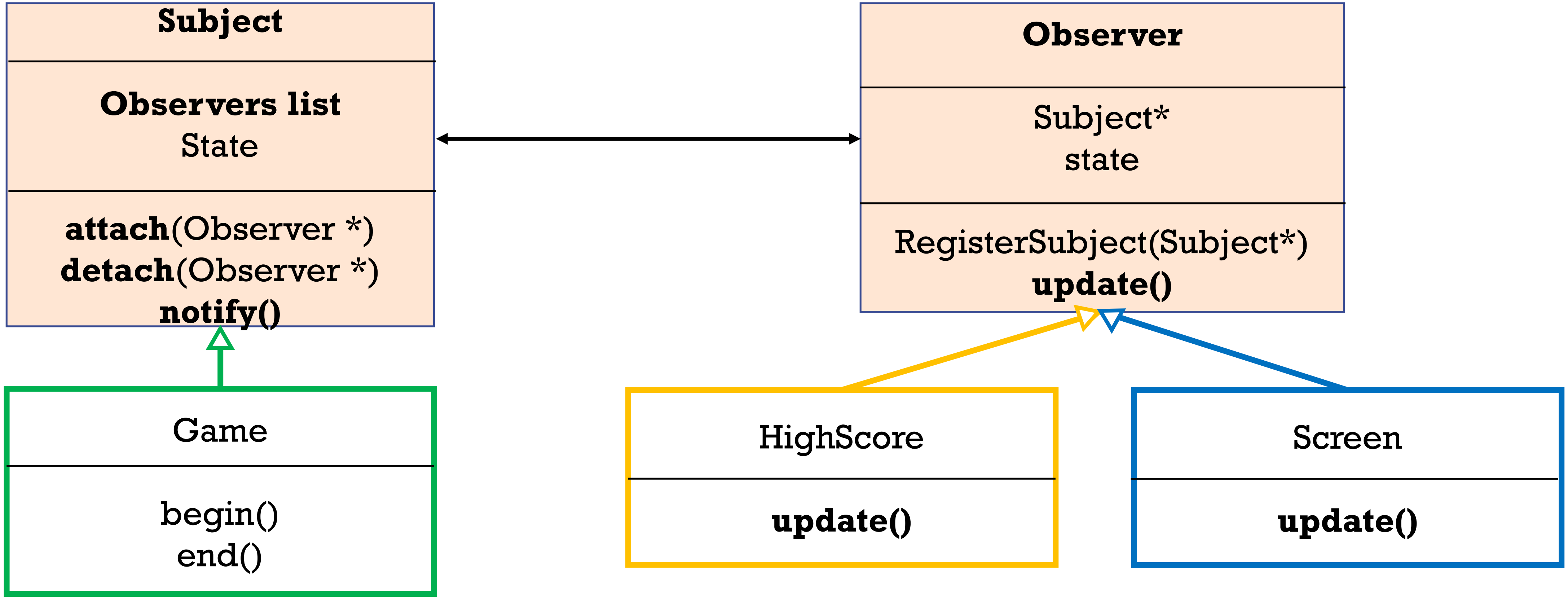
- Use the Observer pattern when:
  - A **change to one subject** requires **changing others**, and you don't know how many objects needs to be changed
  - An **object** needs to **notify other objects** without making any assumptions about what they are (loose coupling!)
- I need the professor to be notified when a student joins his/her class
- I want the display to update when the size of a window is changed
- I need the schedule view to update when the database is changed

Game example

- Game class will notify observers when game begins, and game ends
  - Game begin – HighScore closed, game start screen shown
  - Game end – HighScore displayed, game end screen shown

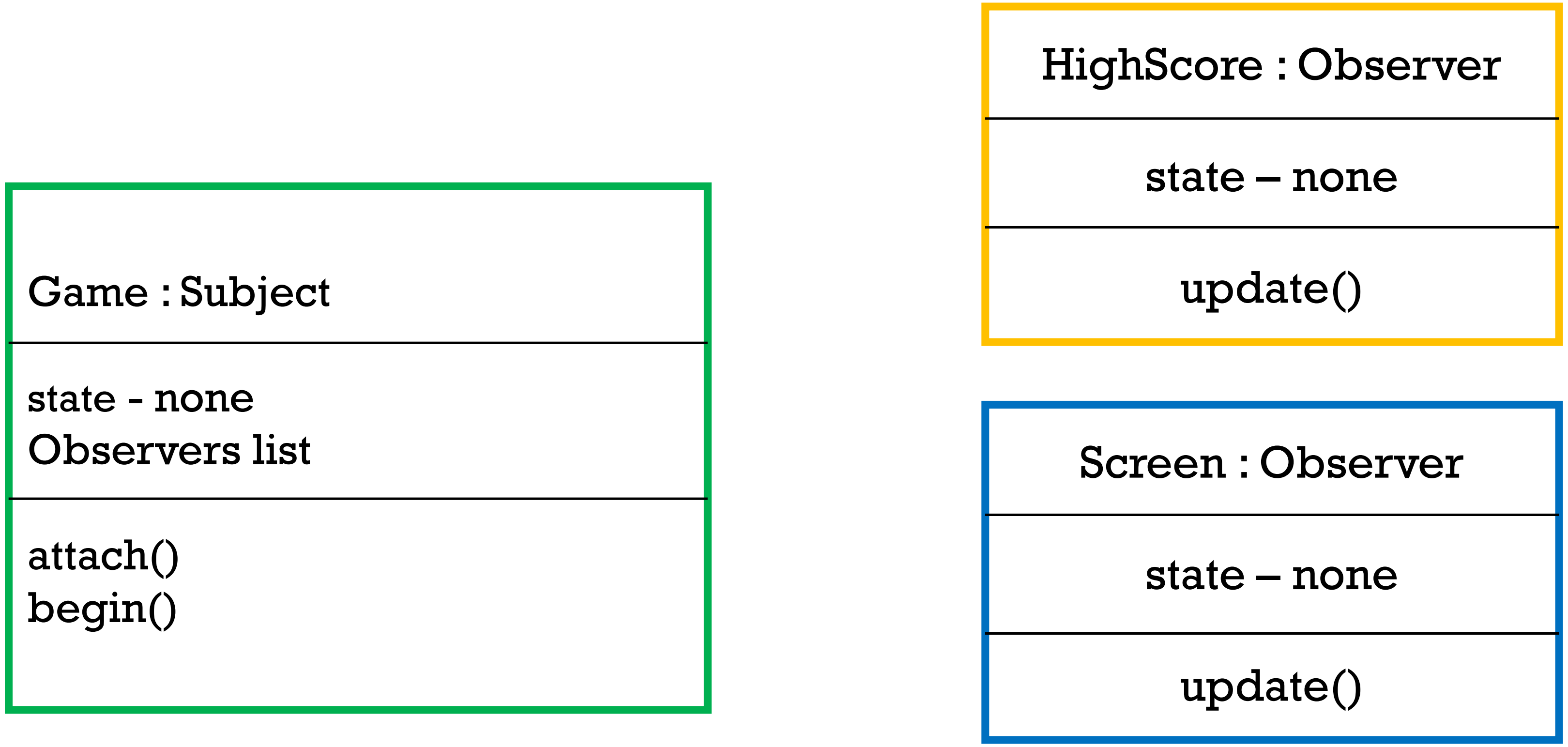
- **Game** class is a **Subject**
  - Subject contains a vector of observer pointers
  - All observers' update() called when subject notify() called
- HighScore and Screen class are **Observers**
  - All observers have an update() function
  - Perform own logic when this update function called by subject

Game example



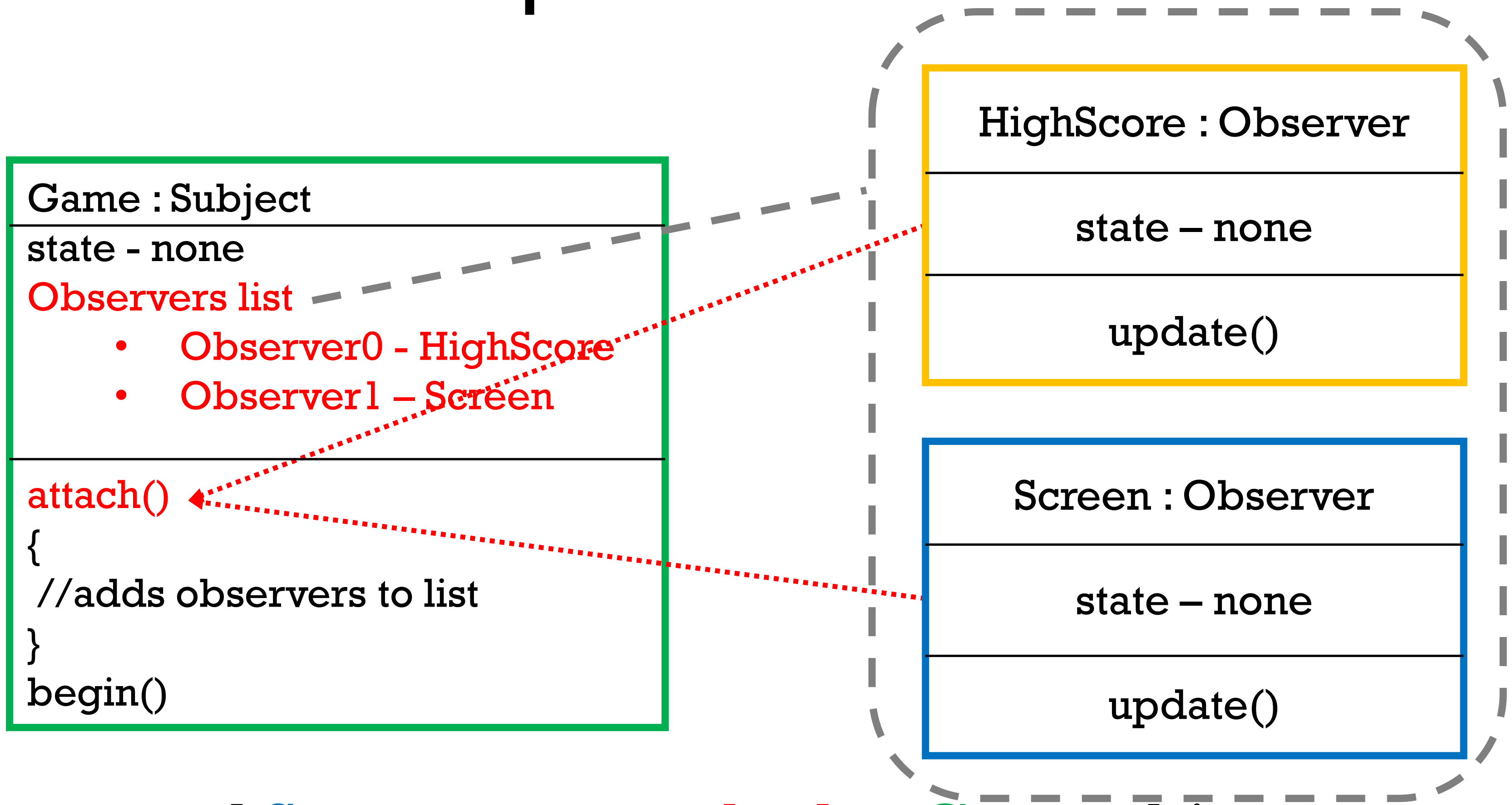
Main
//create HighScore, Screen, Game
//attach Highscore, Screen to Game
//call Game to begin and end

Game flow example



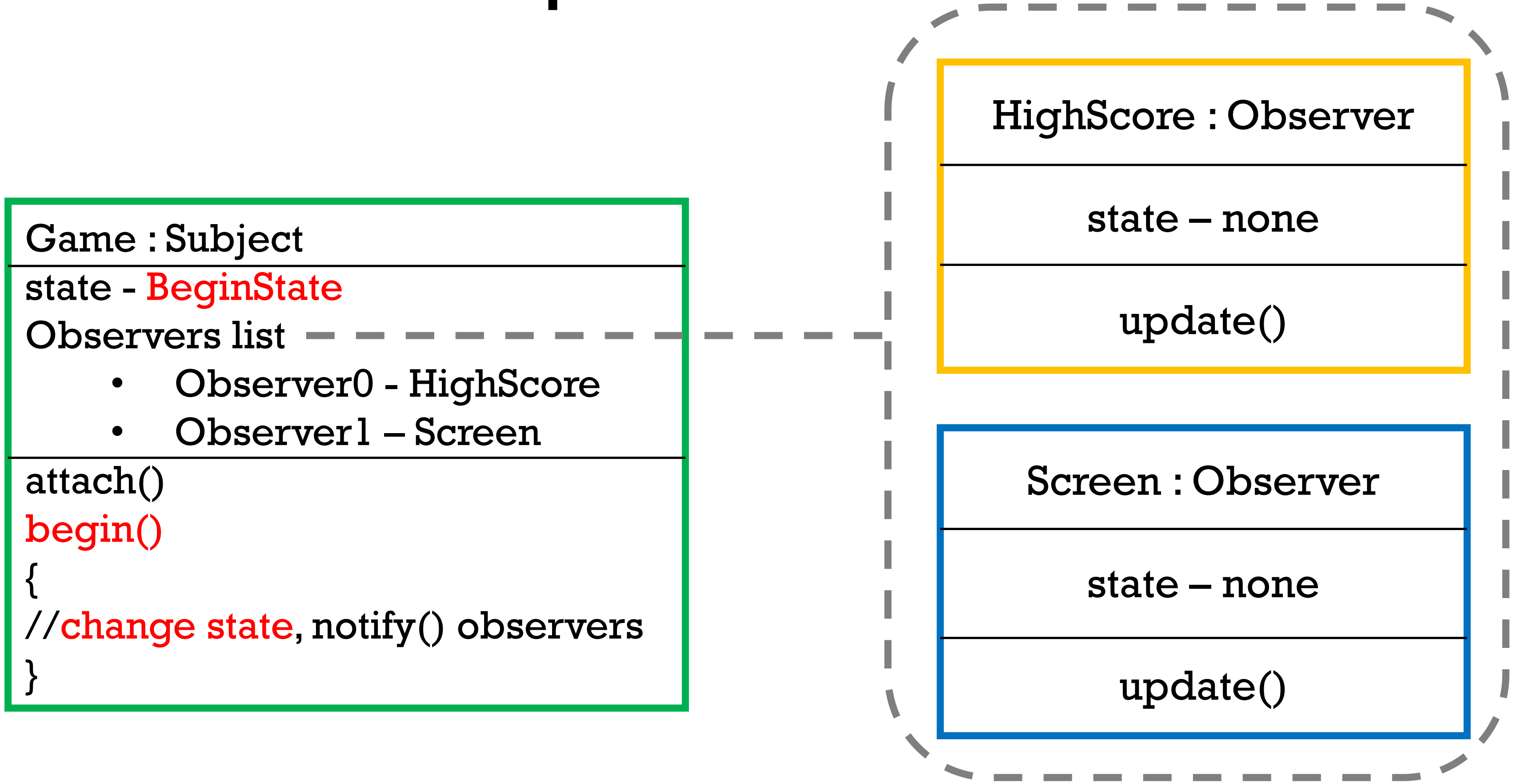
- Main
  - Create Game, HighScore, and Screen objects

Game flow example



- HighScore and Screen are attached to Game object
  - Adds them to the observers list
  - Game doesn't know they're HighScore and Screen objects
  - Game sees them as the **abstract Observer type**

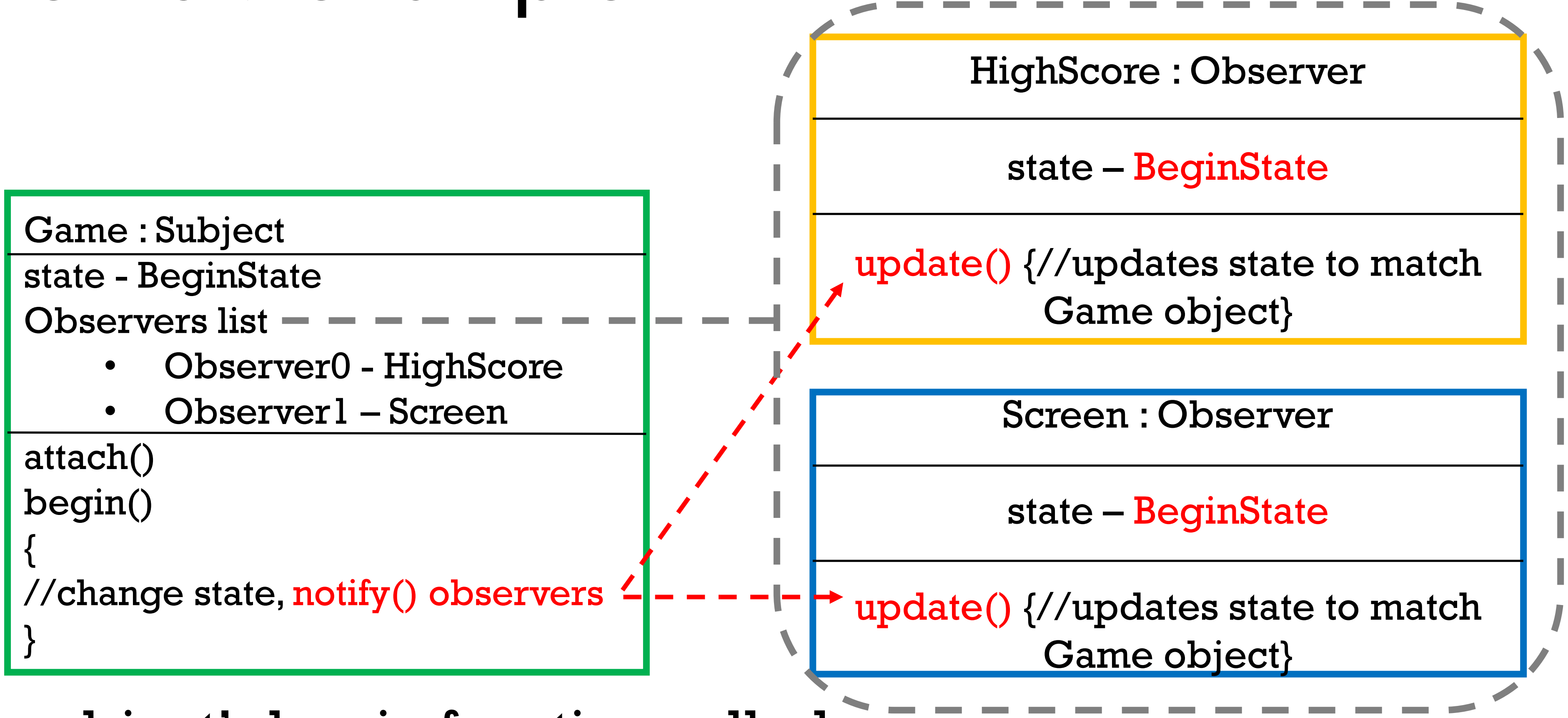
Game flow example



- Game object's begin function called
  - Change the game's state to BeginState

Game flow example

SubjectObserver.cpp



- Game object's begin function called
  - Change the game's state to BeginState
  - Notify all observers in list that something has changed
  - Observers all query the game object to get its current state
  - Sets internal state to match game object's BeginState