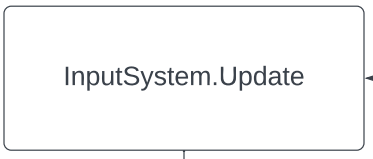
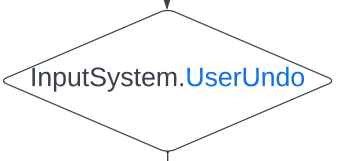
Undo in Big Blue Is You

Update Loop

The game model updates the Input system first, then either calls Undo on all of the other systems, or calls Update on them and pushes a timestamp to the undo stack if updating changed any entities.





Which action is taken depends on whether the user pressed Undo, as determined by the Input system.

No

Yes

```
foreach (System sys in systems)
{
   IEnumerable<Entity> changed = sys.Update(gameTime);
   changedEntities.AddRange(changed);
}

if (changedEntities.Count > 0)
{
   undoStack.Push(gameTime.TotalGameTIme);
}
```

Some of the systems might only be updated in the Draw part of the game loop. In this model those systems are not affected by the user pressing Undo.

```
TimeSpan targetTime = UndoStack.Pop();
foreach (System sys in systems)
{
   foreach (Entity e in sys.Undo(targetTime))
   {
      changedEntities.Add(e);
   }
}
```

```
foreach (Entity e in changedEntities)
{
   foreach (System s in systems)
   {
      // Let each system decide
      // whether to start/keep
      // tracking the changed
      // entity.
      s.InspectEntity(e);
   }
}
```

Classes and Methods

The System class is the common base class for all parents. It keeps a stack of `Tuple<TimeSpan, Func<Entity>>`. Each item in the stack is a timestamp and an action to take to roll a specific change back to that timestamp. System classes should use RegisterUndo to push to this stack and make their actions undoable.

`Func<TResult>` is a delegate type defined by .NET. A delegate is a reference to a method (or a lambda, as in the `Update` example). A `Func<TResult>` instance must return `TResult`.

The action required by RegisterUndo must return an `Entity` so the game model can mark that entity as changed and let all systems inspect it after an undo is performed.

```
Class System
```

```
+ undoActions : Stack<Tuple<TimeSpan, Func<Entity>>> =
new()
```

```
+ Undo(target: TimeSpan) : IEnumerable<Entity>
+ Update(GameTime gameTime)
# RegisterUndo(GameTime gameTime, Func<Entity>
undoAction) : void
```

```
public IEnumerable<Entity> Undo(TimeSpan target)
{
   // Executes this system's undo actions until none are
   // left or the top action
   // occurred before the given time target.
   // Returns an enumerable of changed entities.

   HashSet<Entity> changedEntities = new HashSet<Entity>();
   while (UndoActions.TryPeek(out Tuple<TimeSpan, Func<Entity>>
undoAction) && undoAction.Item1 >= target)
   {
      Entity changedEntity = UndoActions.Pop().Item2.Invoke();
      if (changedEntity != null)
         changedEntities.Add(changedEntity);
   }
   return changedEntities;
}
```

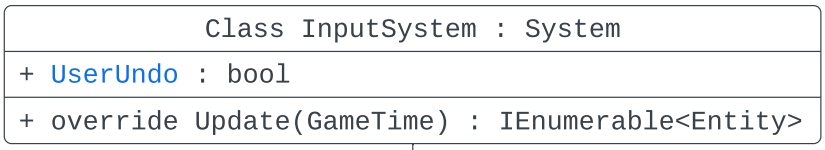
```
public IEnumerable<Entity> Update(GameTime gameTime)
{
   // Do stuff unique to the system, and use RegisterUndo
   // to allow those actions to be undone.

   // This method is abstract in ECS.System, but the following
   // is an example that registers an undo action for changing
   // an entity's position component.
   PositionComponent position =
         entity.GetComponent<PositionComponent>();
   position.X = newX;
   position.Y = newY;
   RegisterUndo(gameTime,
      () =>
      {
         // This does not work if Component is a value type
         position.X = oldX;
         position.Y = oldY;
         return entity;
      }
   );

   yield return entity;
}
```

```
public void RegisterUndo(GameTime gameTime, Func<Entity> undoAction)
{
   Tuple<TimeSpan, Func<Entity>> pair =
         new Tuple<TimeSpan, Func<Entity>>(gameTime.TotalGameTime, undoAction);

   undoActions.Push(pair);
}
```




```
public override IEnumerable<Entity> Update(GameTime gameTIme)
{
   UserUndo = false;

   // Does updates specific to the input system and sets
   // UserUndo to true if any user has pressed their
   // Undo key.
}
```