Unity

Philippines Users Group

May 2013 Meet-up

facebook.com/groups/unitypug

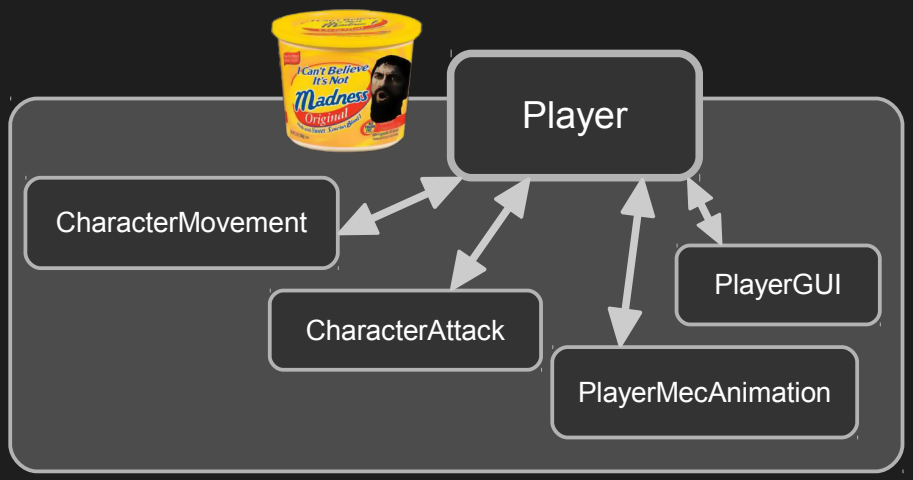
unity3d.org.ph

Design Patterns

Design Patterns

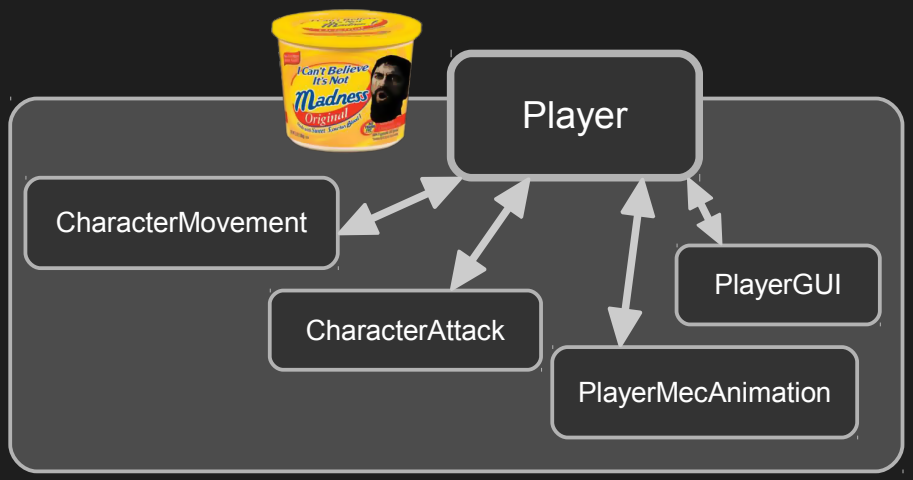
What they are
Why they are useful
When to use them

Remember our set of classes earlier?



Remember our set of classes earlier?

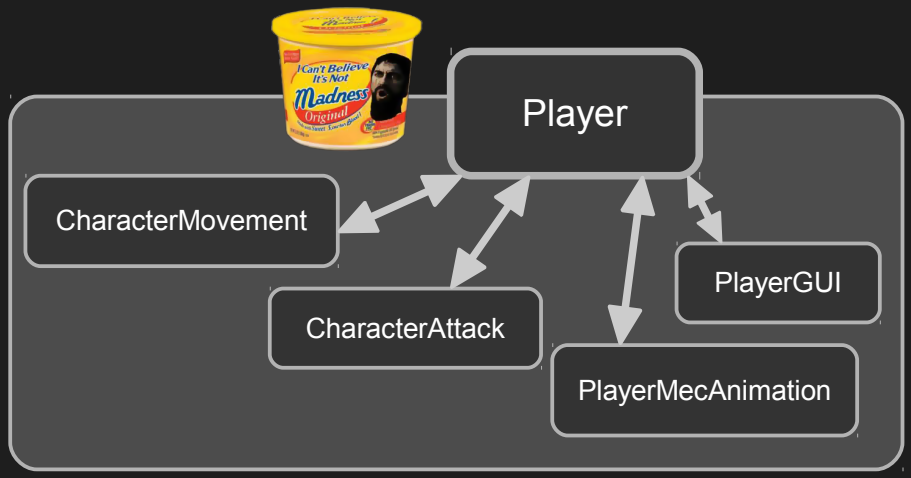
Turns out I'm not the only one using this technique.

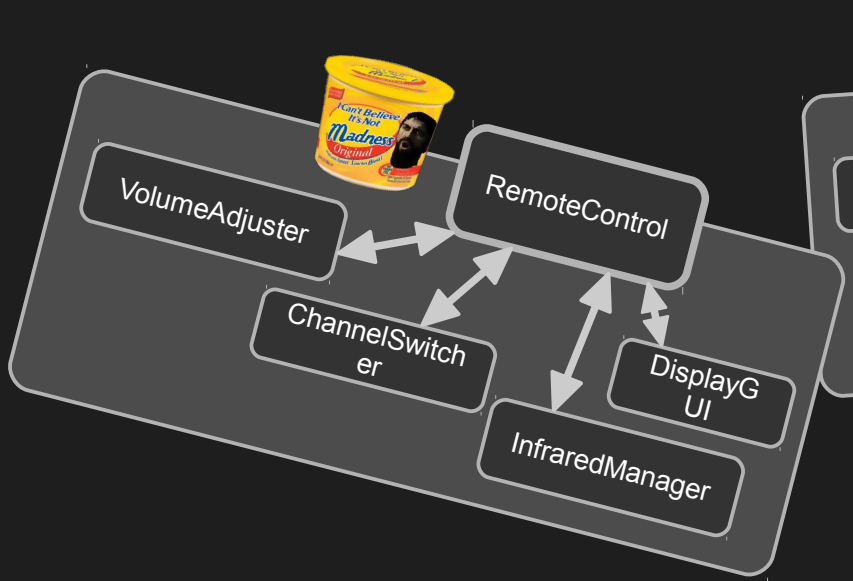


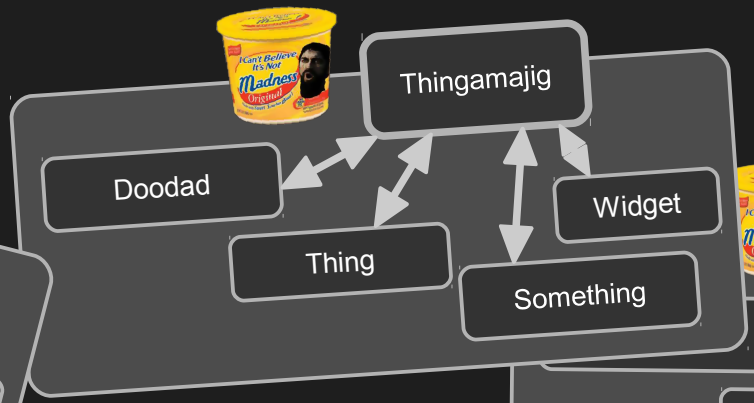
Remember our set of classes earlier?

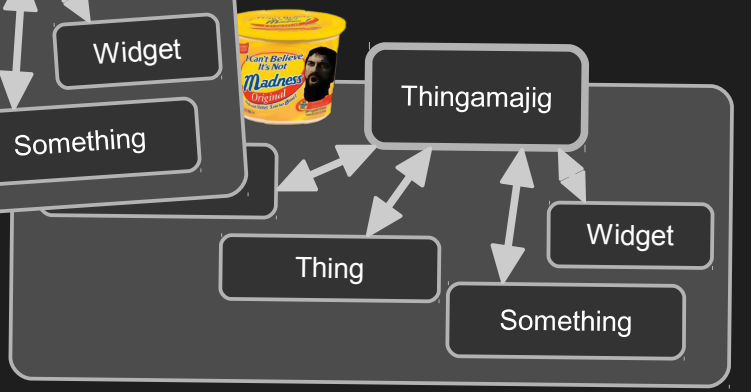
Lots of people are designing their classes this way too.

Turns out I'm not the only one using this technique.







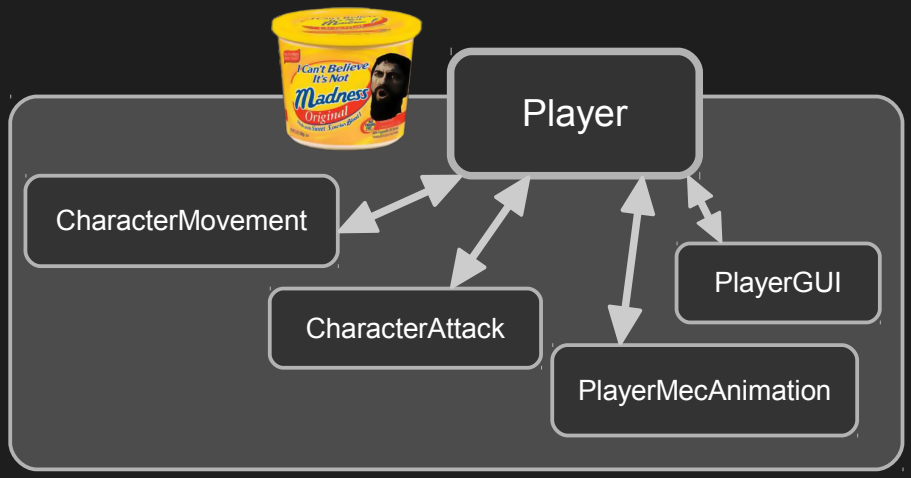


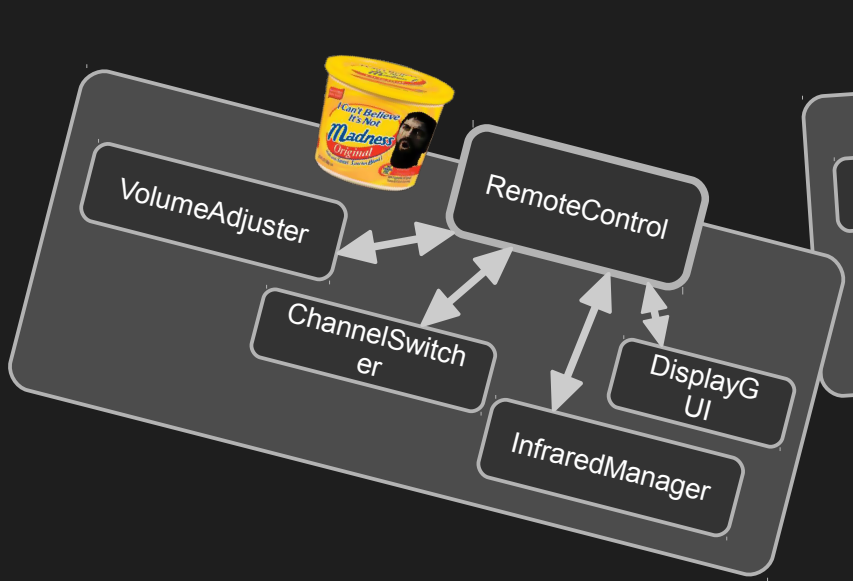
Remember our set of classes earlier?

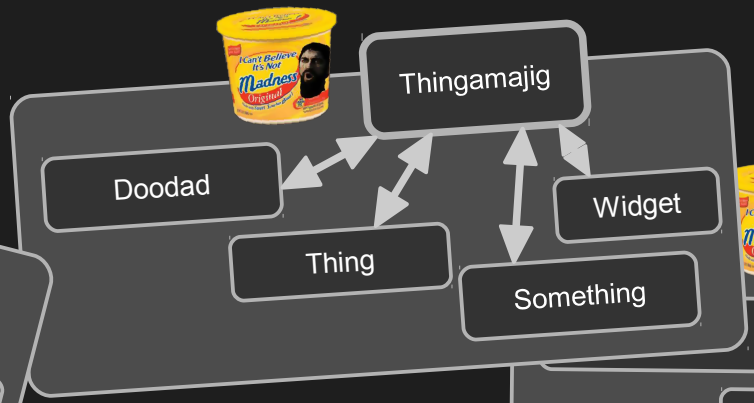
Turns out I'm not the only one using this technique.

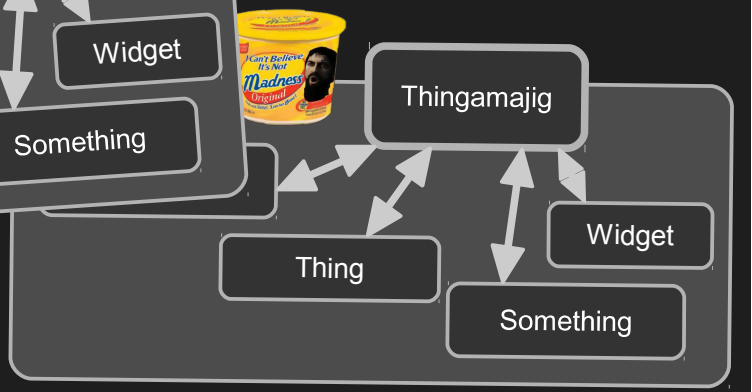
Lots of people are designing their classes this way too.

We didn't copy from each other, we just realized on our own, this was the better way to design classes.



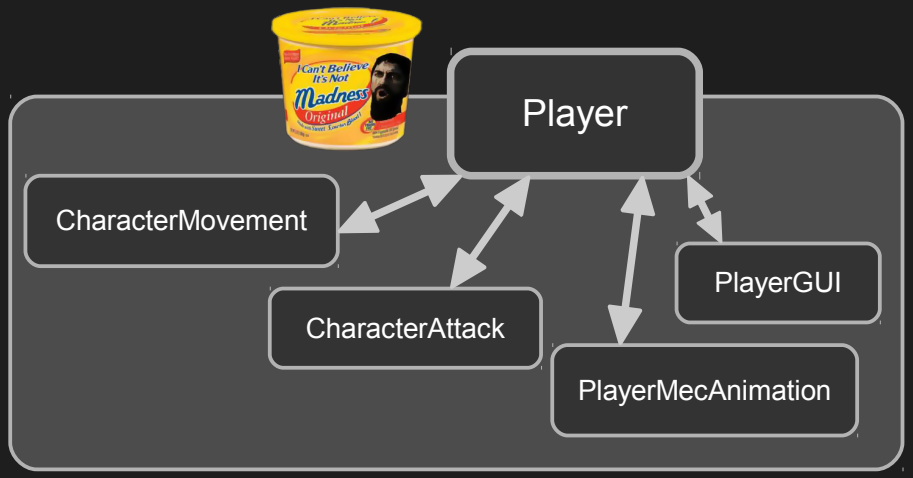


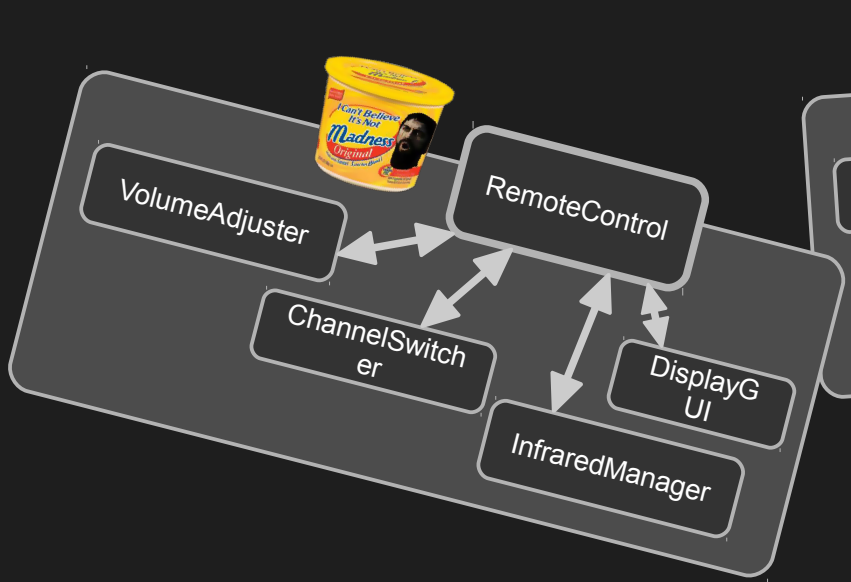


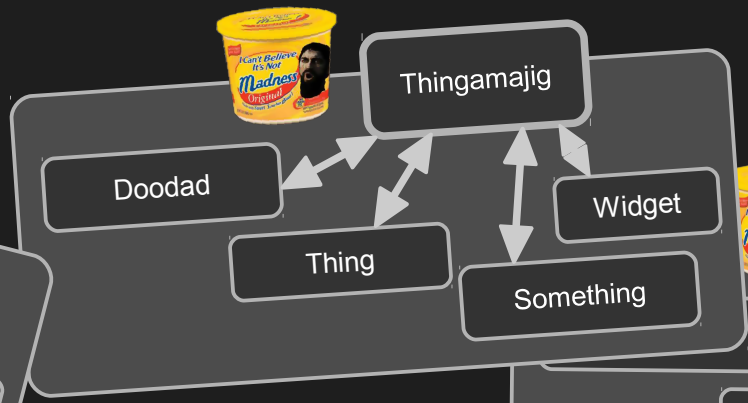


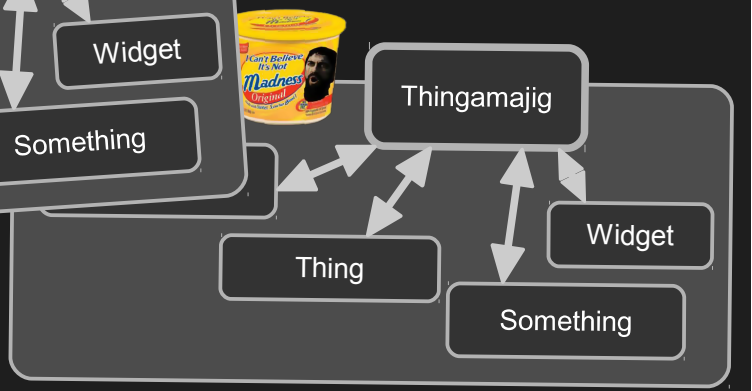
But four guys realized this

"Gang of Four"



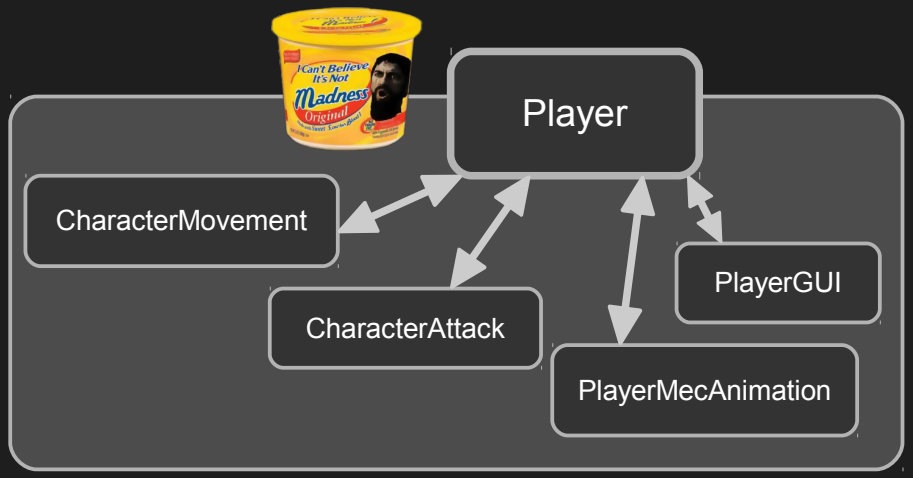




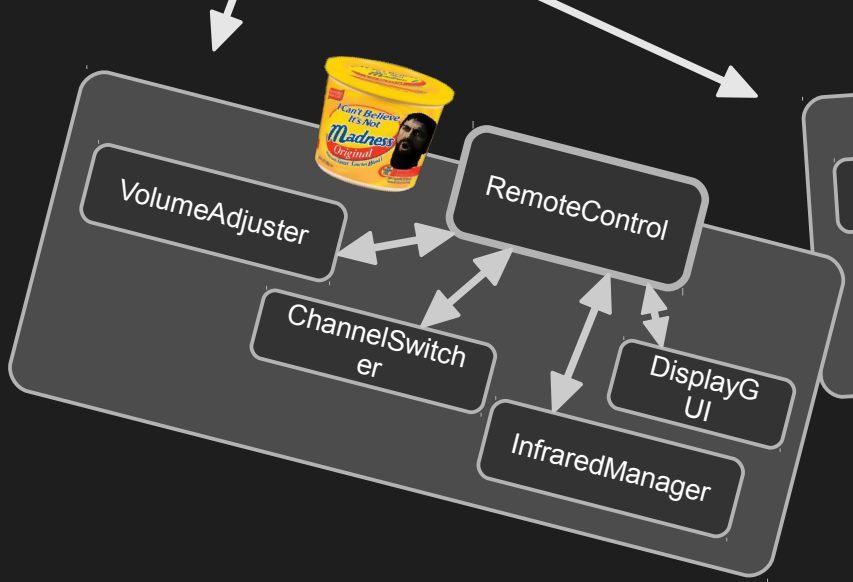


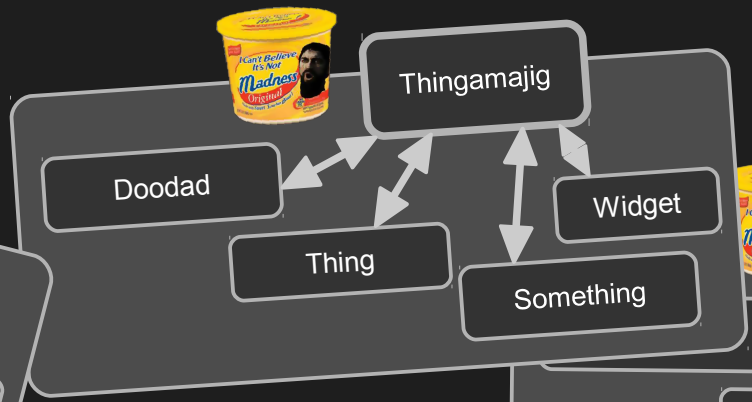
But four guys realized this

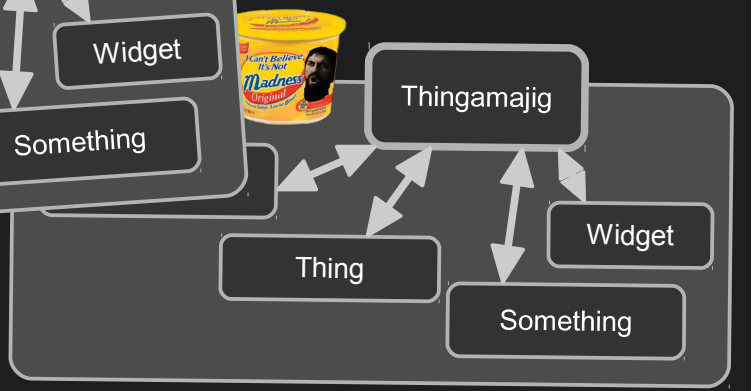
"Gang of Four"



ZOMG they're the same!



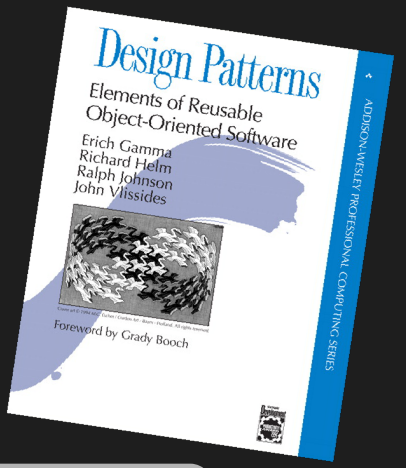


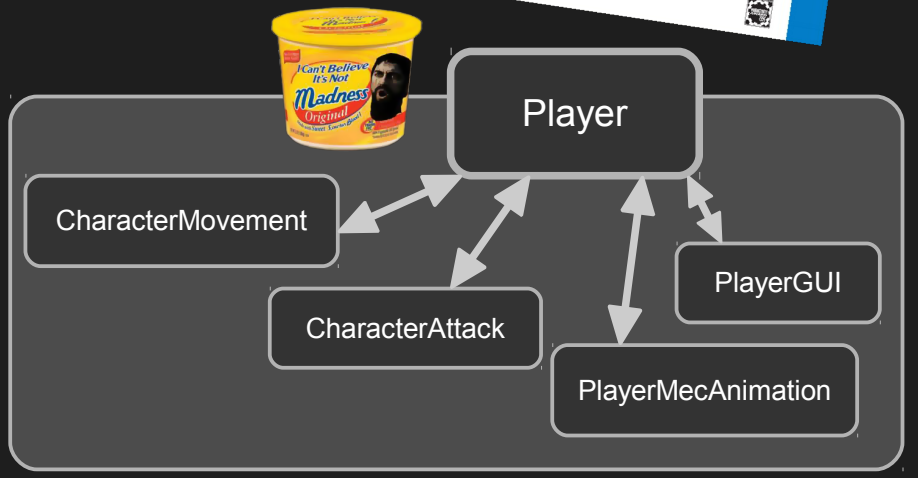


But four guys realized this

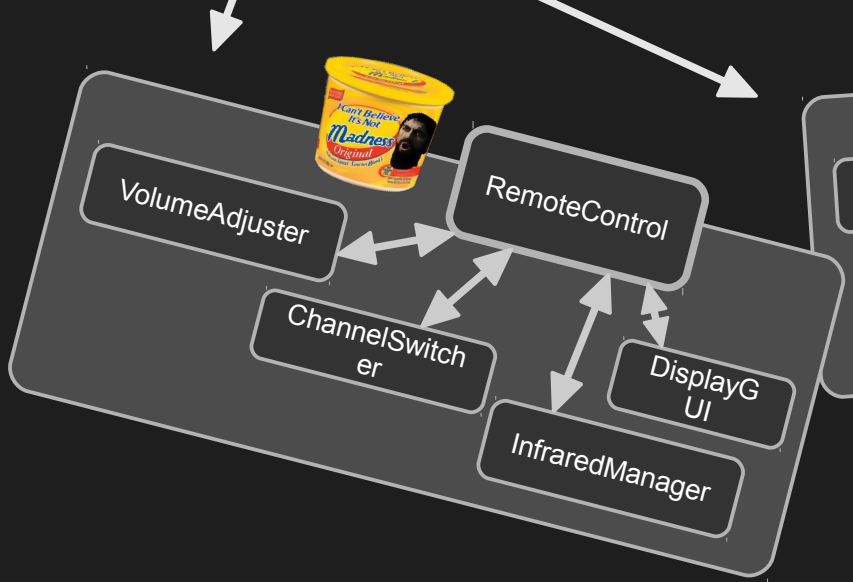
"Gang of Four"

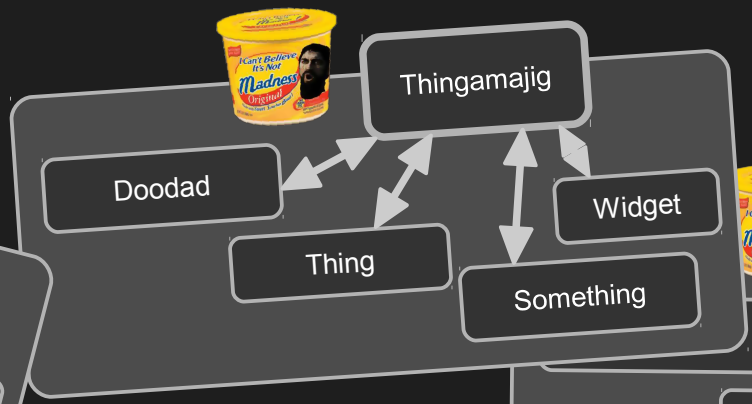
And they created a compendium about class designs that people used a lot

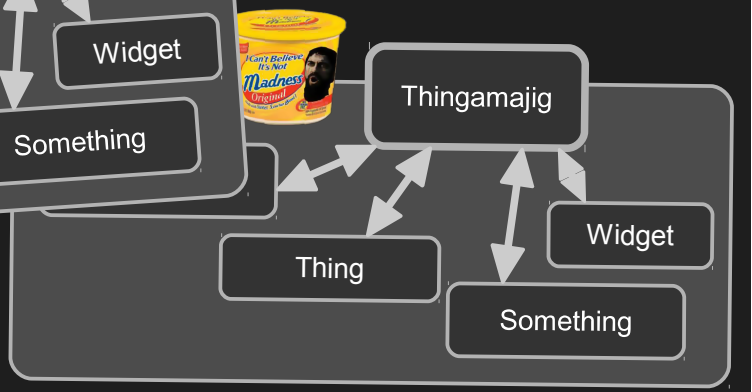




ZOMG they're the same!



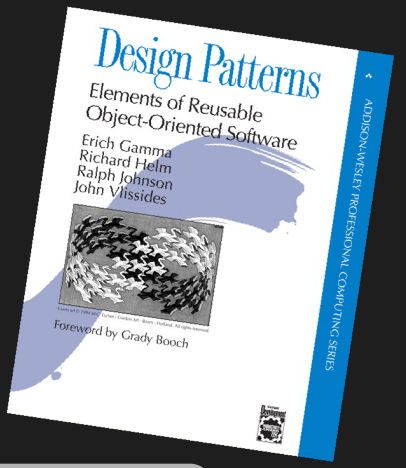




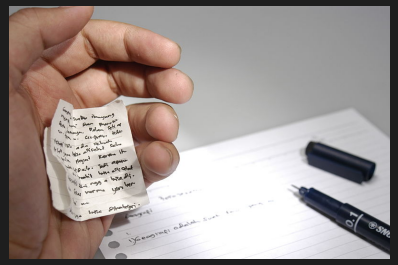
But four guys realized this

"Gang of Four"

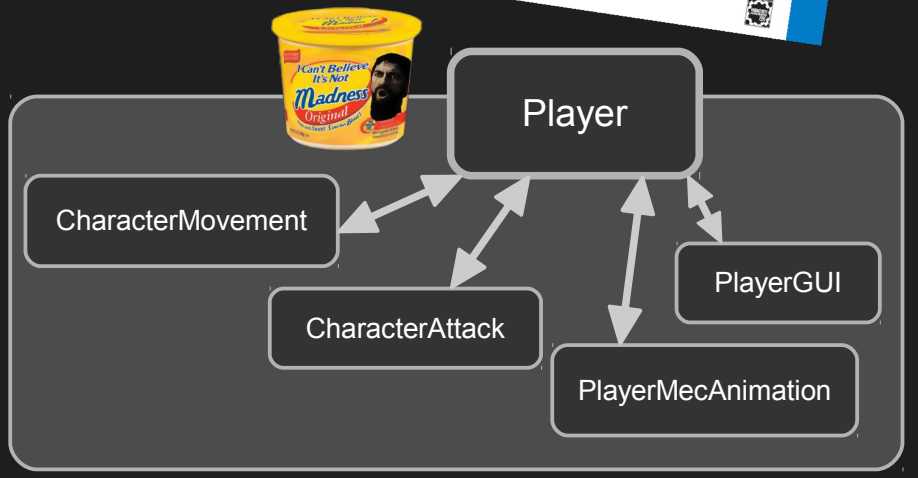
And they created a compendium about class designs that people used a lot

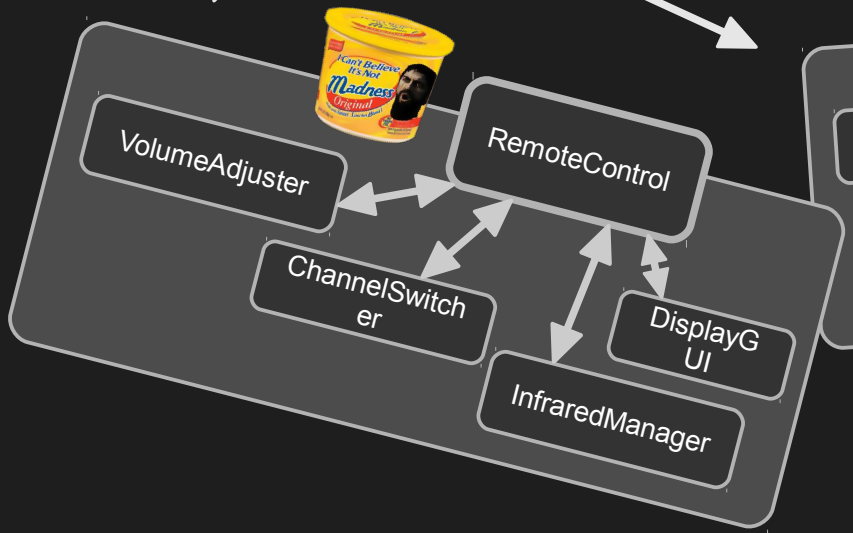


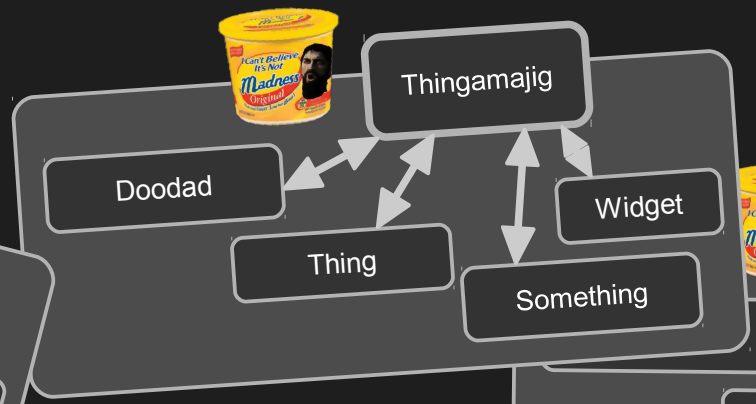
So now we can share cheat sheets with each other!

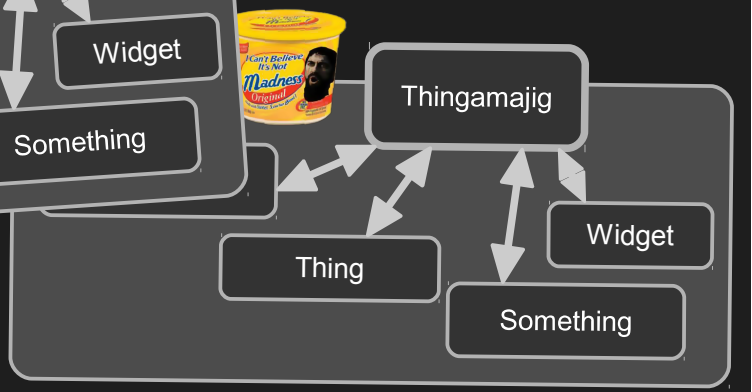


ZOMG they're the same!









They called those designs...

They called those designs...

Design Patterns

They called those designs...

Design Patterns

I'm beginning to see an emerging pattern here...

Here's an example pattern:

Here's an example pattern:

Observer Pattern

Here's an example pattern:

Observer Pattern

Some people may call this:

- Listener Pattern
- Subscriber Pattern

Here's an example pattern:

Observer Pattern

Some people may call this:
- Listener Pattern
- Subscriber Pattern

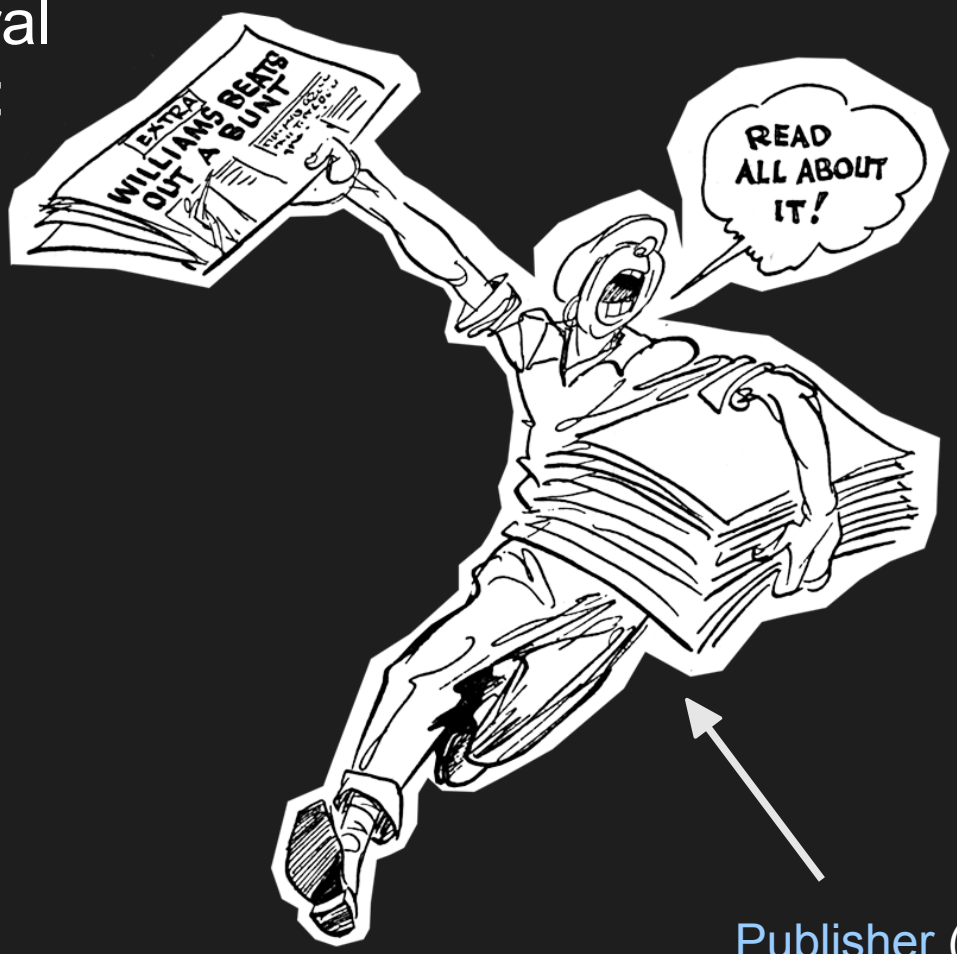
General idea:

Here's an example pattern:

Observer Pattern

Some people may call this:
- Listener Pattern
- Subscriber Pattern

General idea:



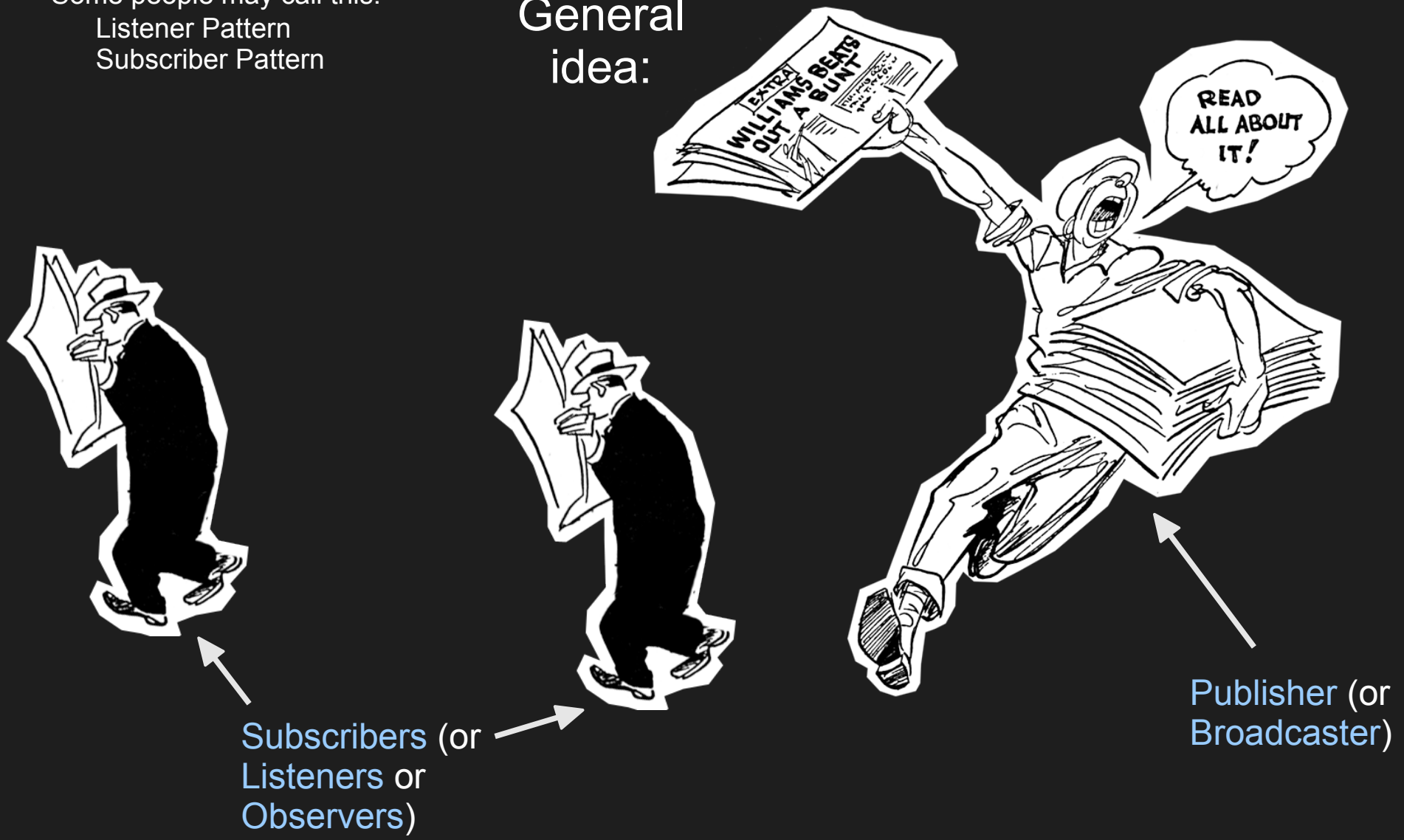
Publisher (or Broadcaster)

Here's an example pattern:

Observer Pattern

Some people may call this:
- Listener Pattern
- Subscriber Pattern

General idea:

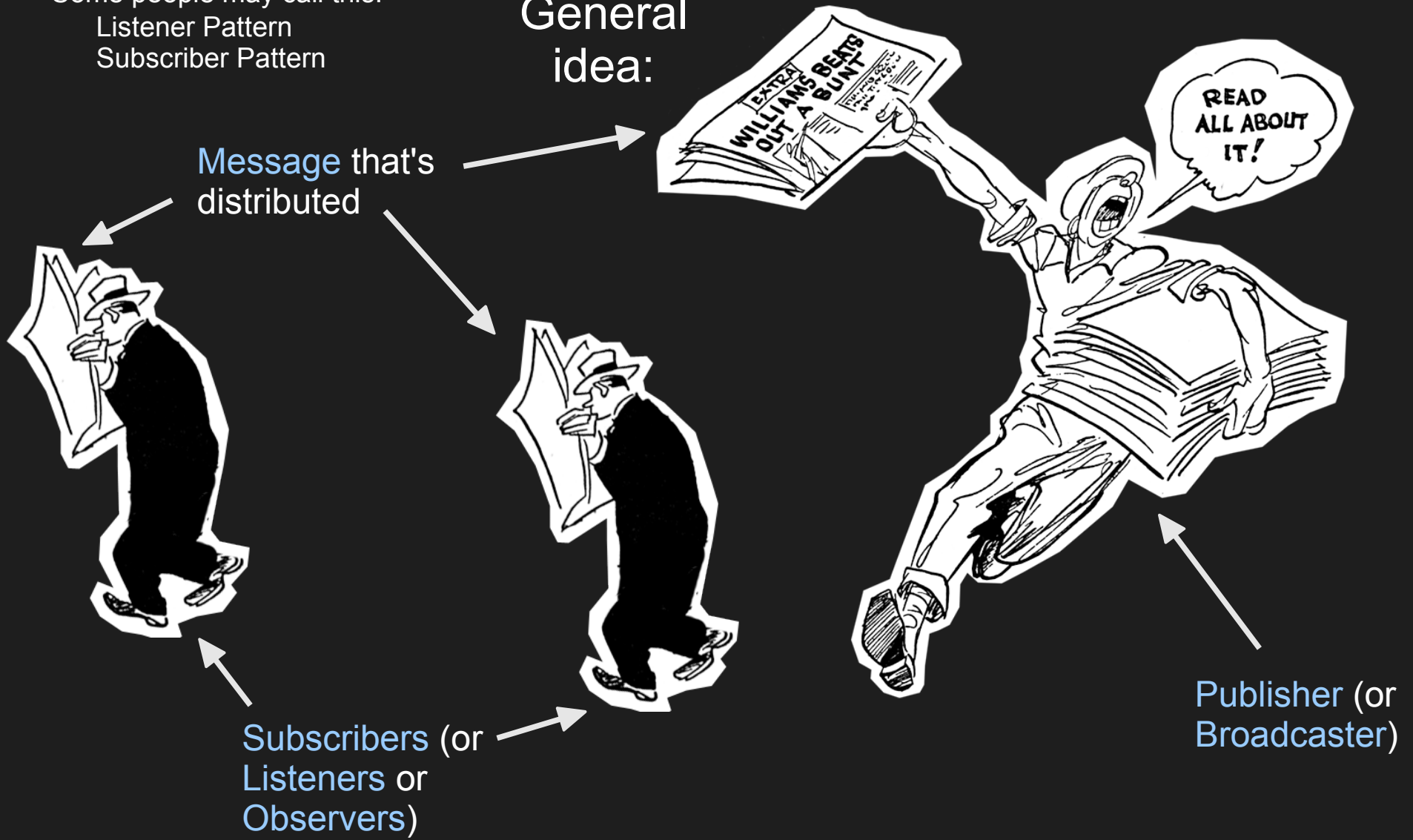


Here's an example pattern:

Observer Pattern

Some people may call this:
- Listener Pattern
- Subscriber Pattern

General idea:



Message that's distributed

Subscribers (or Listeners or Observers)

Publisher (or Broadcaster)

Ok, sure. But how is this Observer Pattern useful to game programming?

Ok, sure. But how is this Observer Pattern useful to game programming?

Answer: “Depends.”

Ok, sure. But how is this Observer Pattern useful to game programming?

Answer: “Depends.”

You can actually use it for many things.

Ok, sure. But how is this

Observer Pattern useful to game programming?

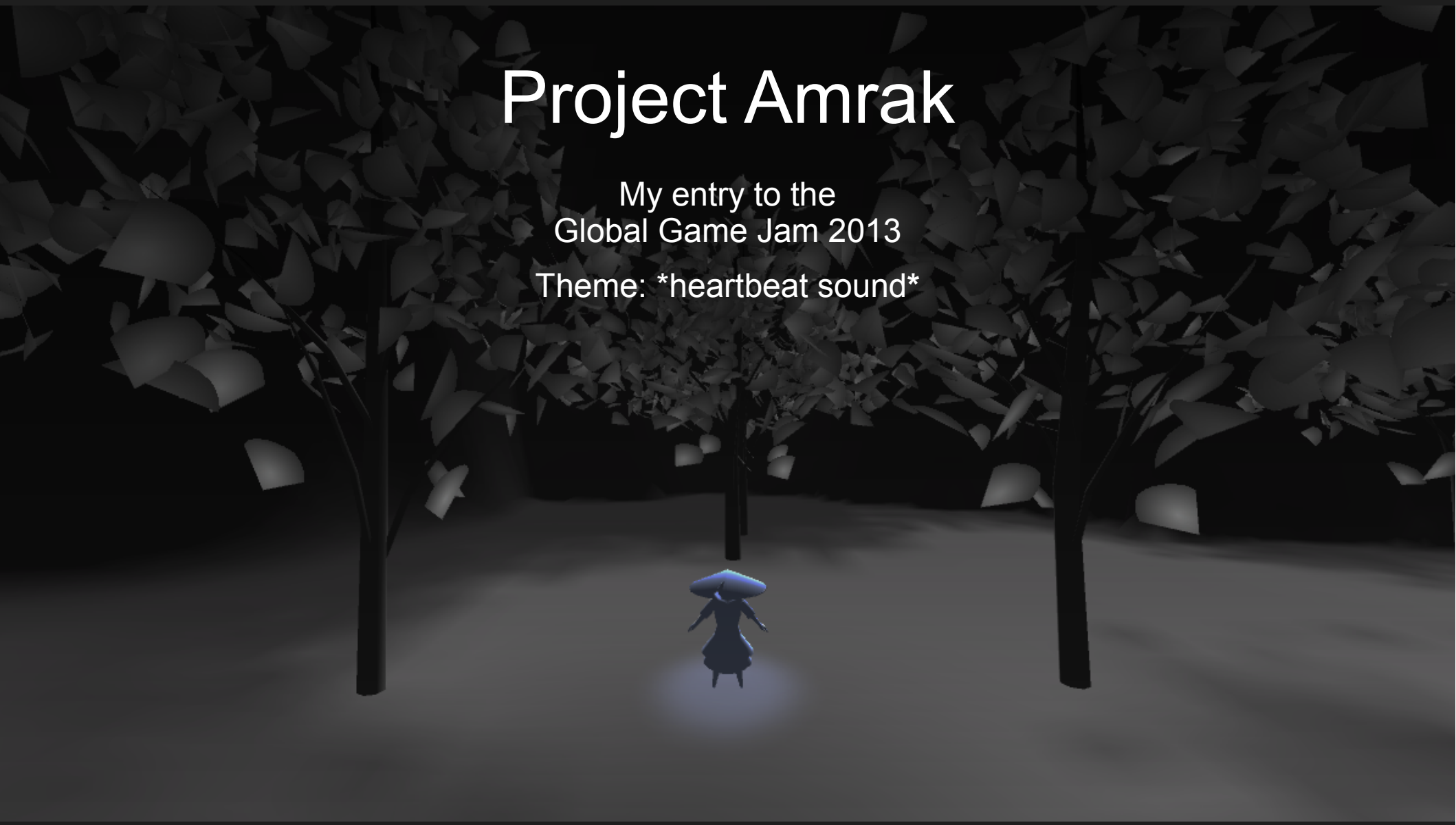
Answer: “Depends.”

You can actually use it for many things.

Here's one example...

Project Amrak


Project Amrak
My entry to the
Global Game Jam 2013


Project Amrak
My entry to the
Global Game Jam 2013
Theme: *heartbeat sound*


Project Amrak
My entry to the
Global Game Jam 2013
Theme: *heartbeat sound*
Uses Observer Pattern

Project Amrak

My entry to the
Global Game Jam 2013

Theme: *heartbeat sound*

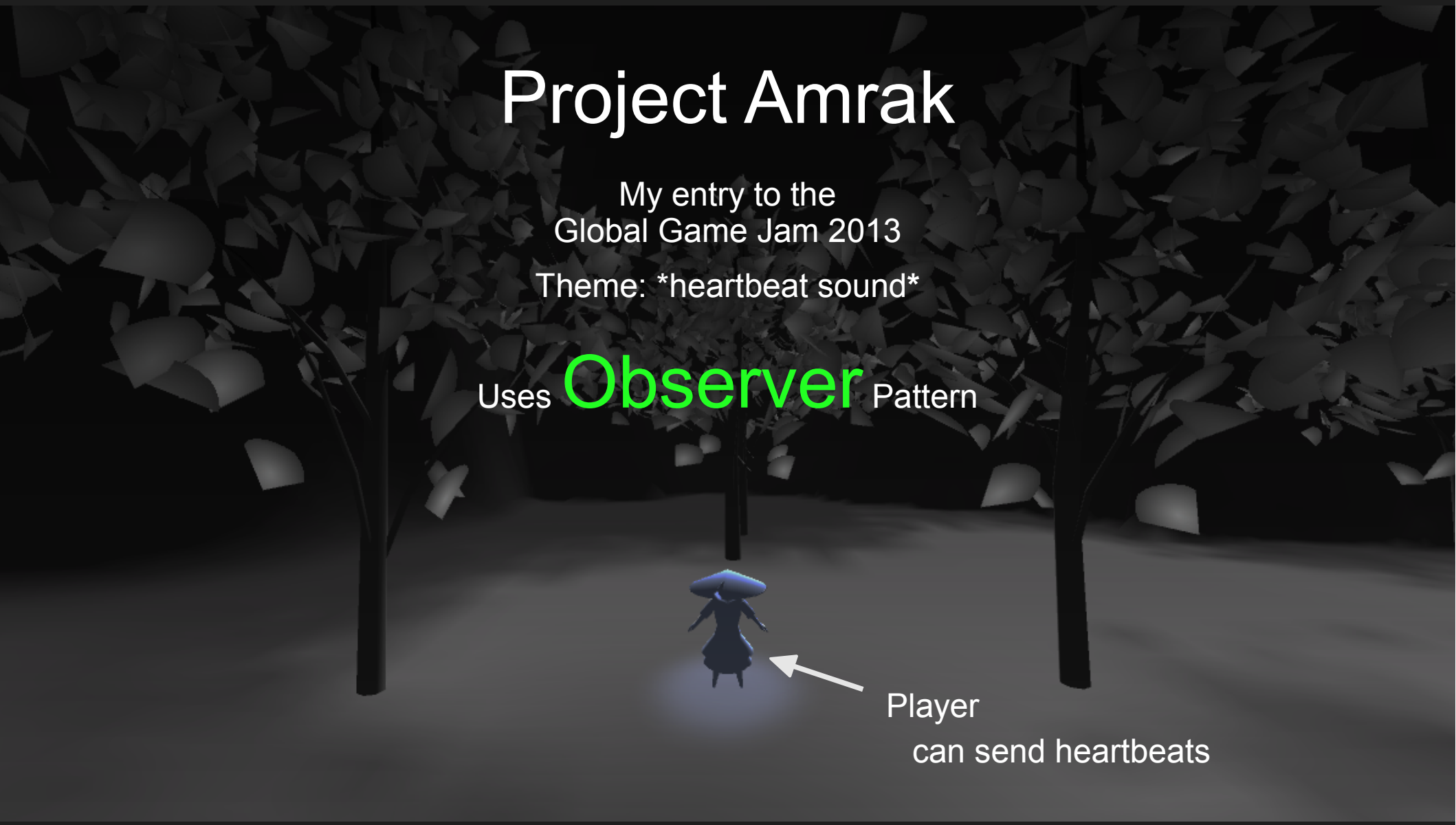
Uses **Observer** Pattern

Project Amrak

My entry to the
Global Game Jam 2013

Theme: *heartbeat sound*

Uses **Observer** Pattern

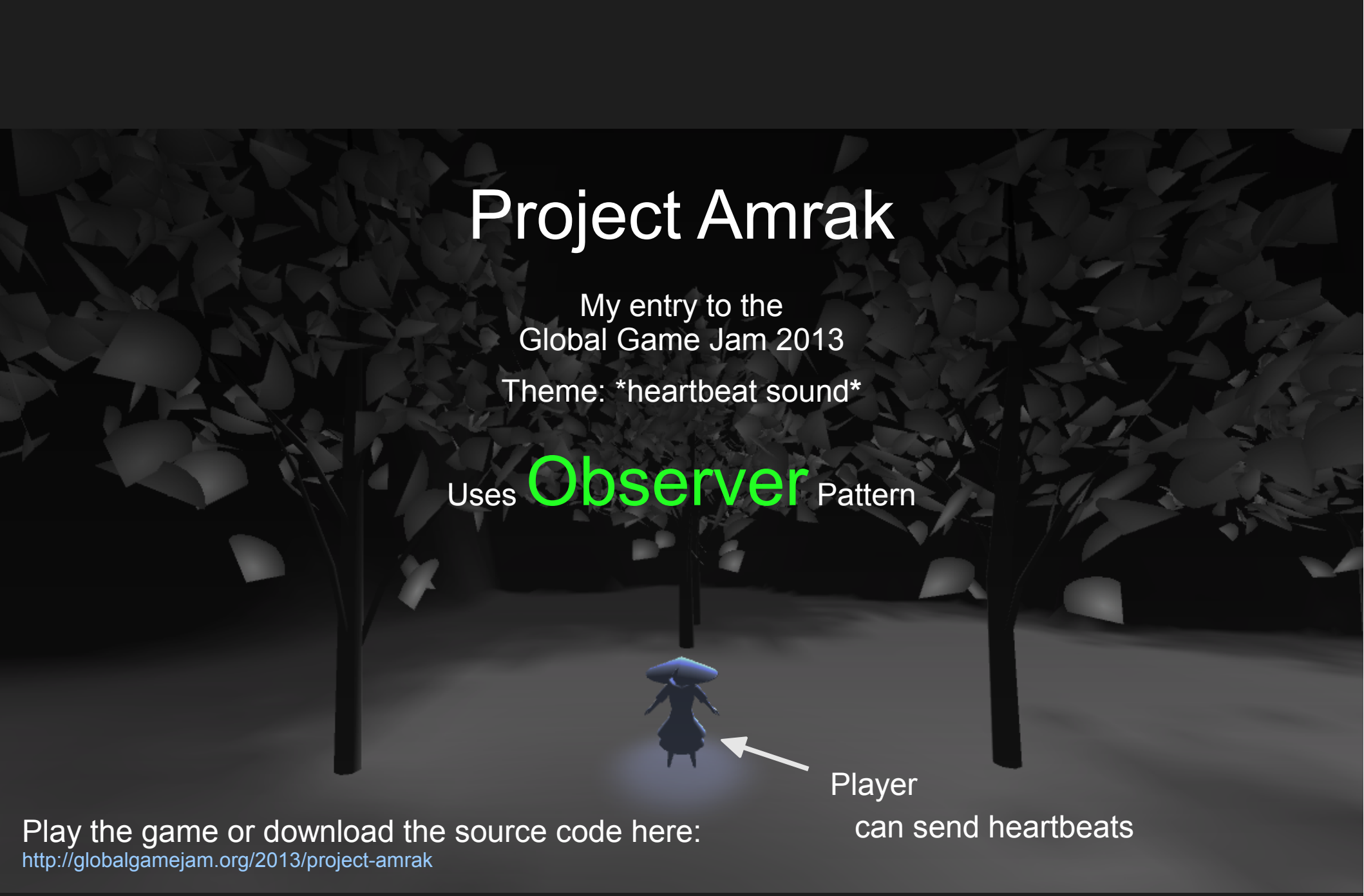


Project Amrak

My entry to the
Global Game Jam 2013

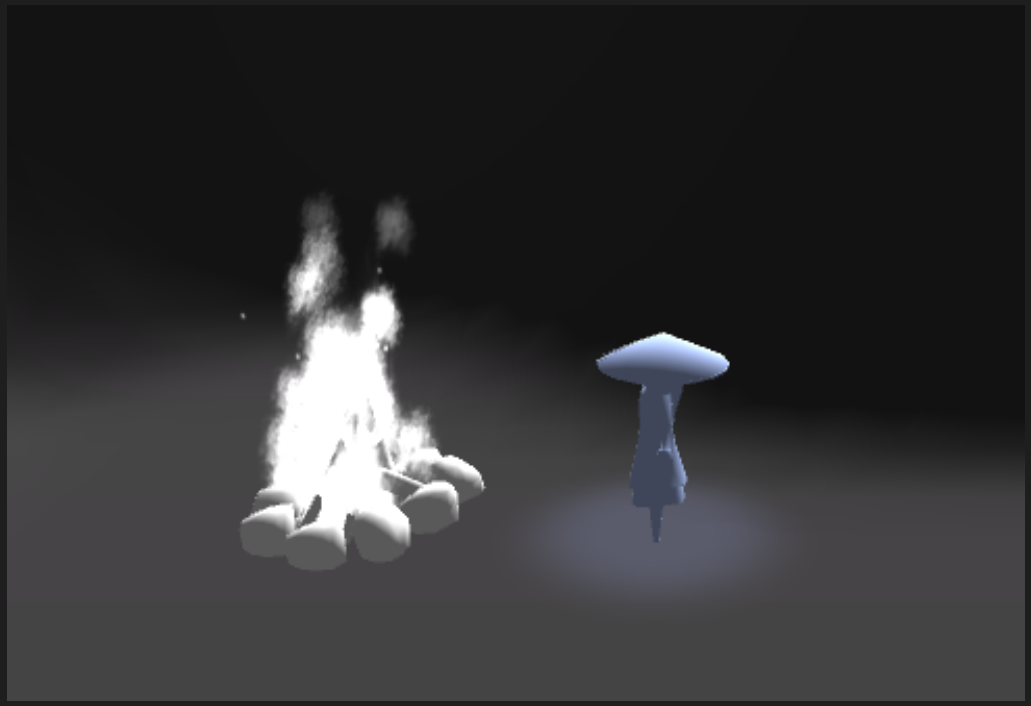
Theme: *heartbeat sound*

Uses **Observer** Pattern

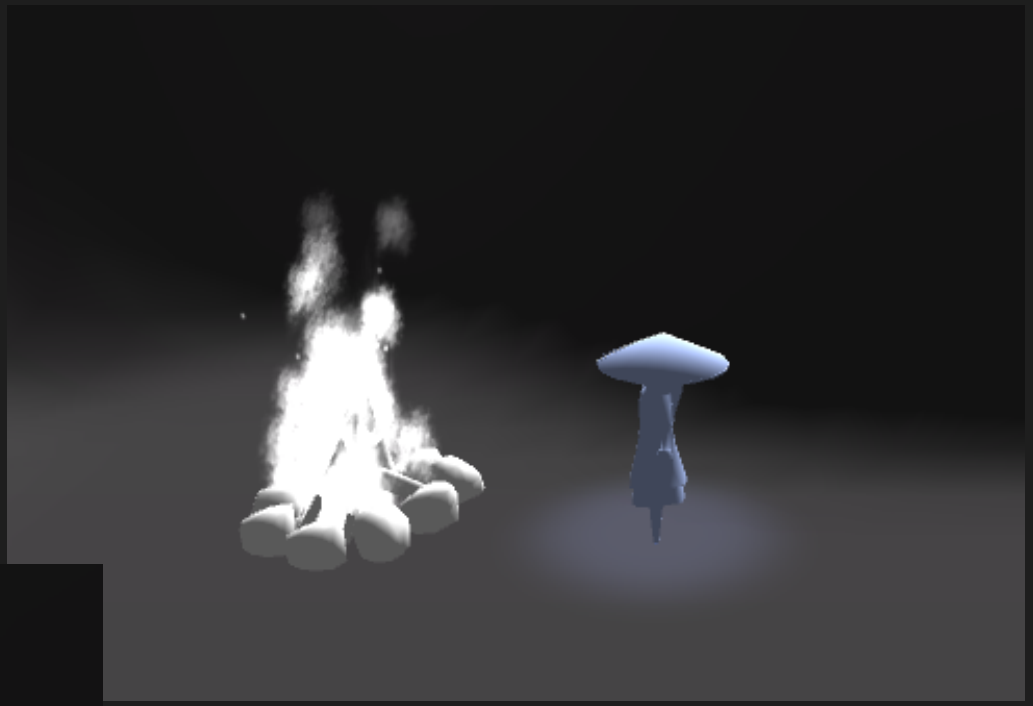


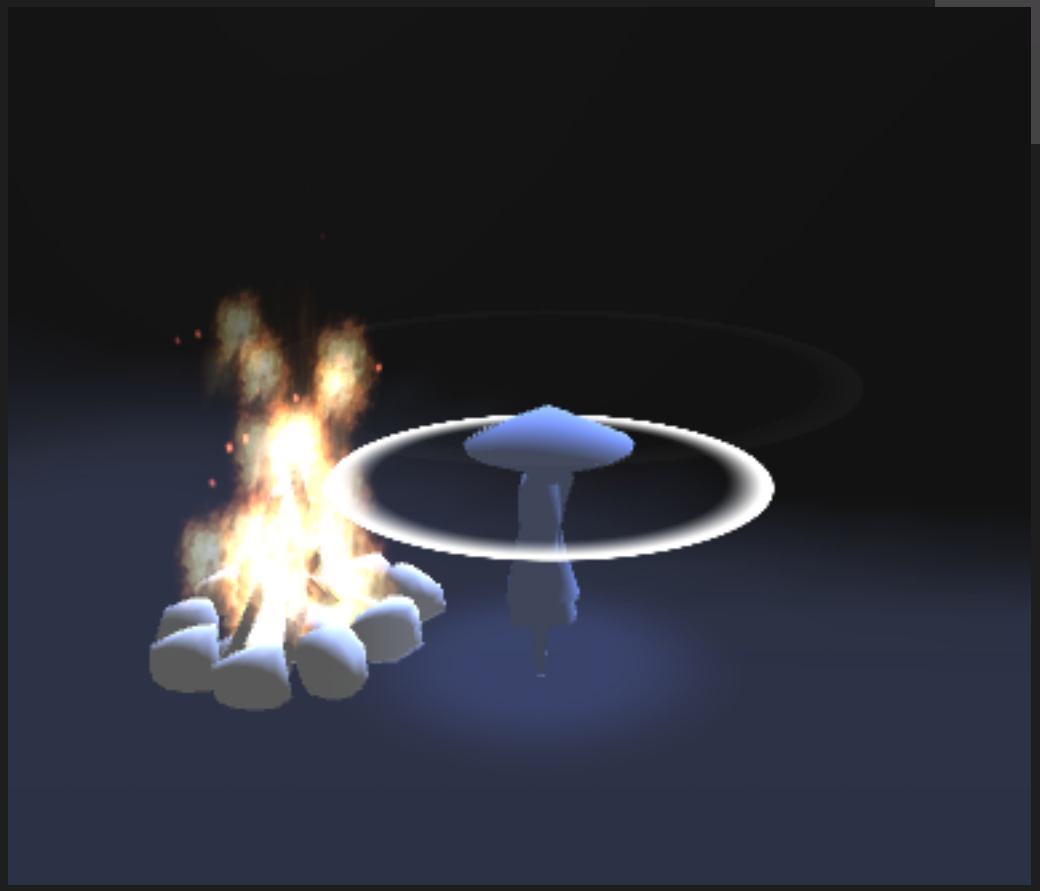
Play the game or download the source code here:
http://globalgamejam.org/2013/project-amrak

When the player sends a heartbeat sound to an object...



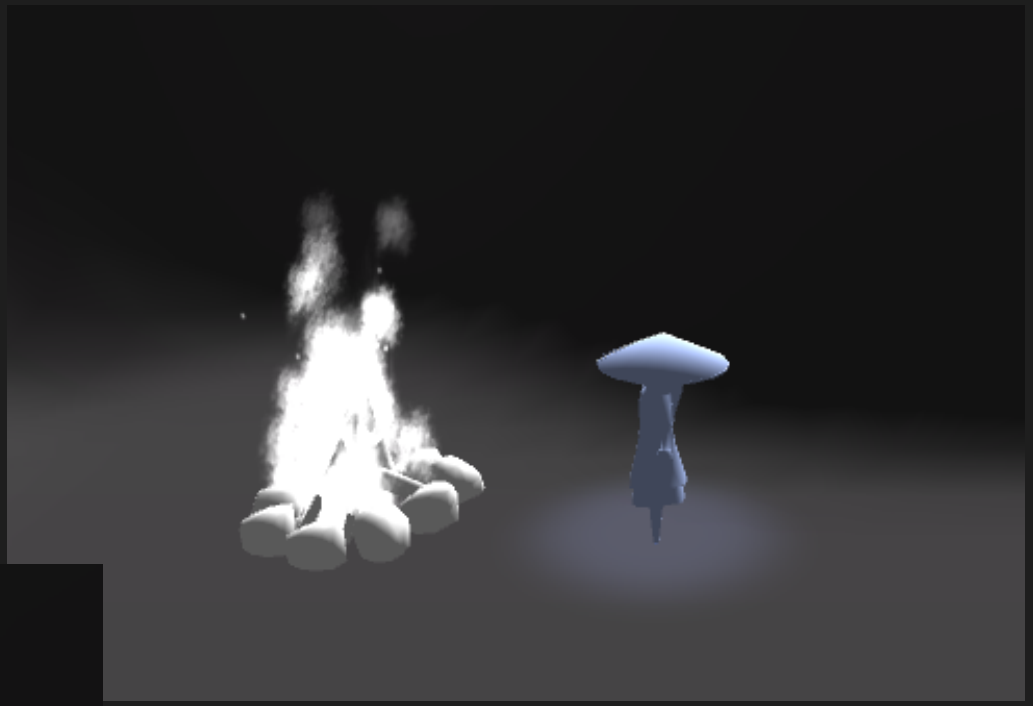
When the player sends a heartbeat sound to an object...

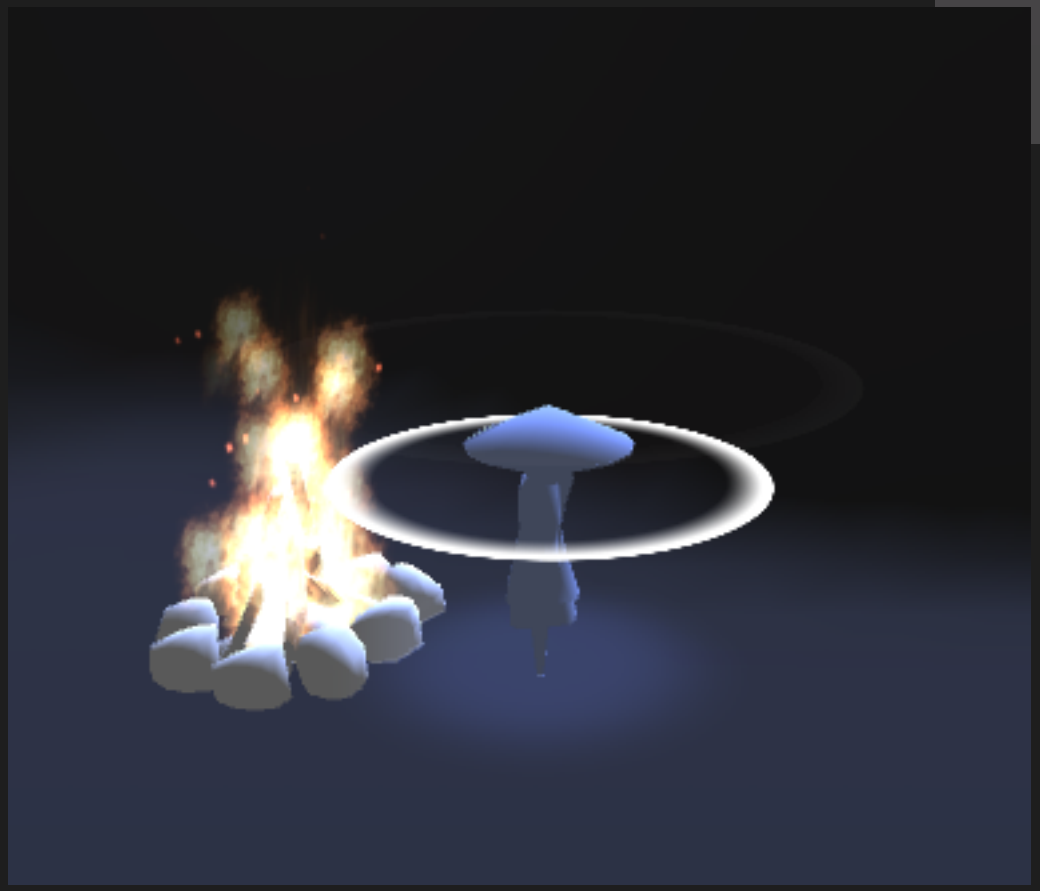




...the object will respond somehow.

When the player sends a heartbeat sound to an object...

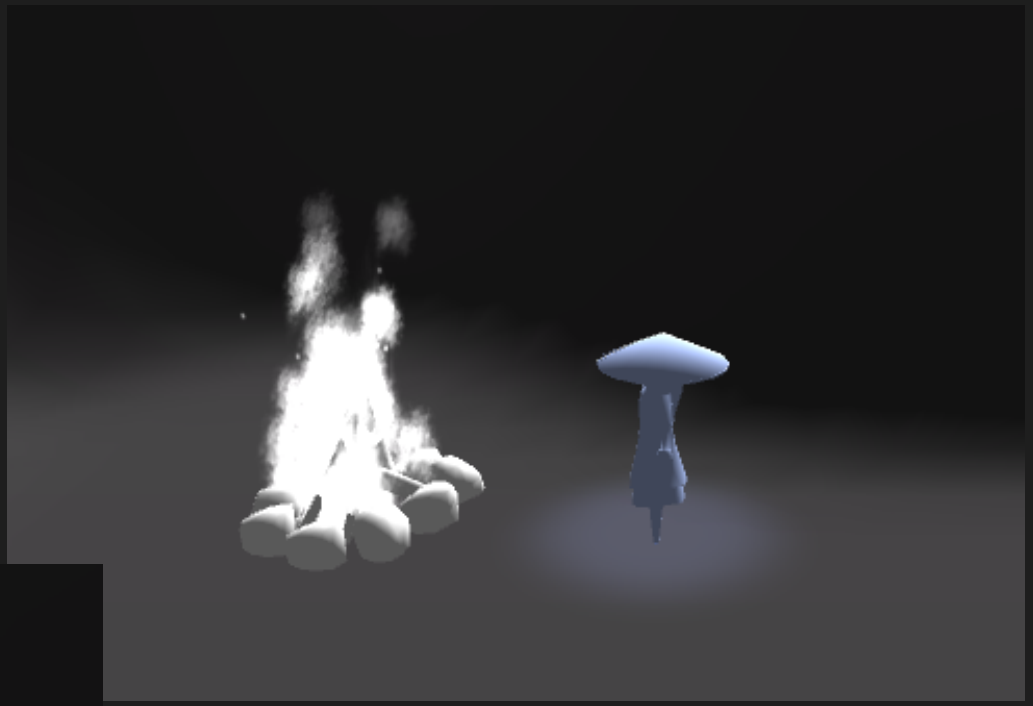


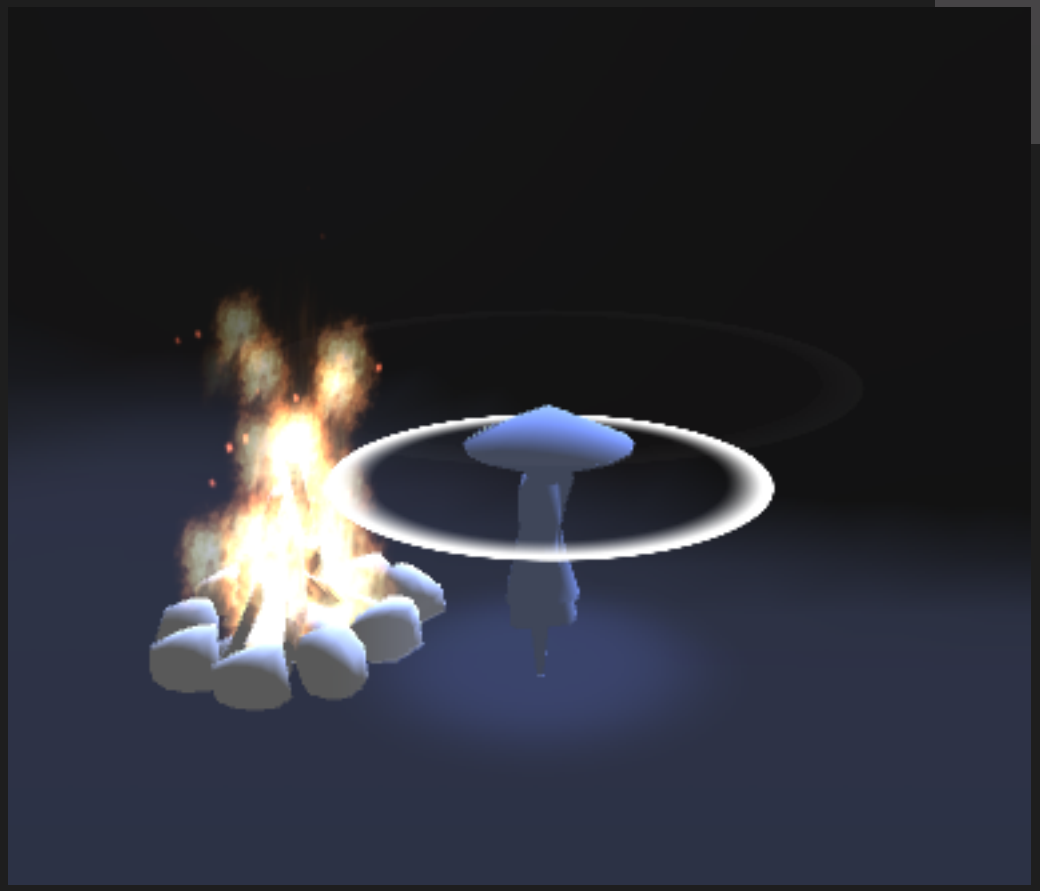


...the object will respond somehow.

In this case, the frozen campfire will start to move.

When the player sends a heartbeat sound to an object...





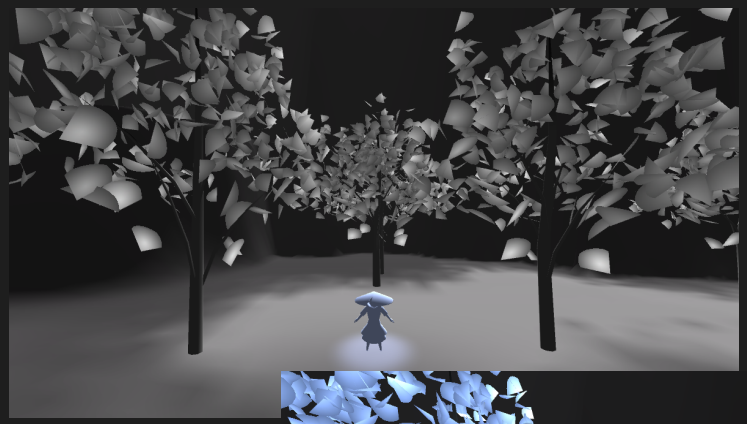
...the object will respond somehow.

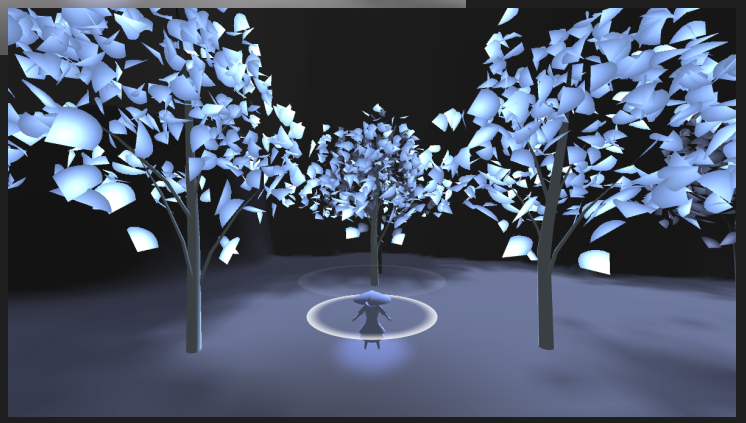
In this case, the frozen campfire will start to move.

The game is centered around solving puzzles this way.

Different objects react to the heartbeat differently.

Different objects react to the heartbeat differently.

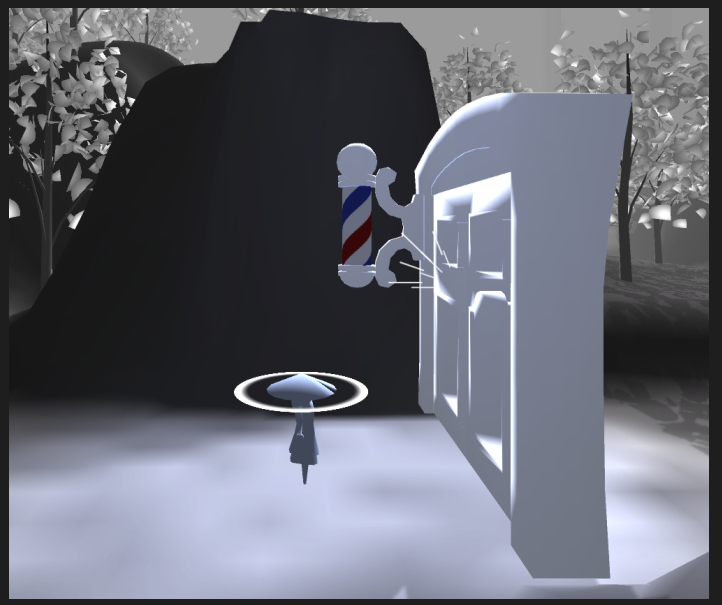


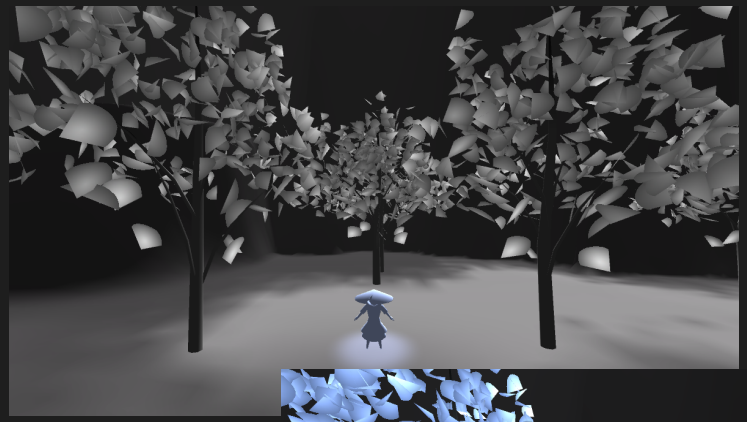


Trees shake and gain color when receiving a heartbeat.

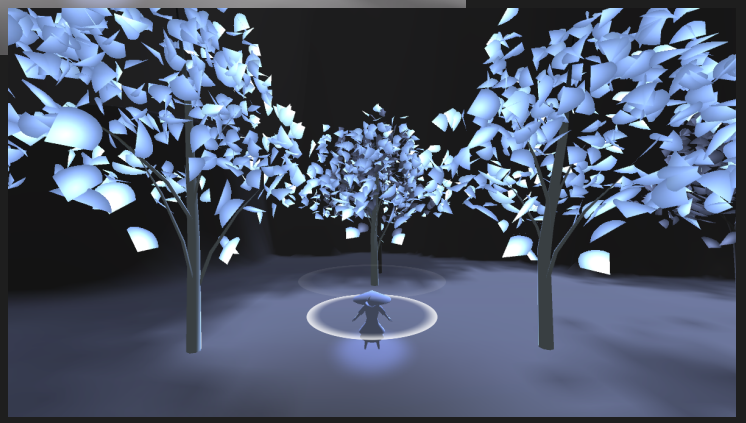
Different objects react to the heartbeat differently.

This door will open only when you time your heartbeat properly.



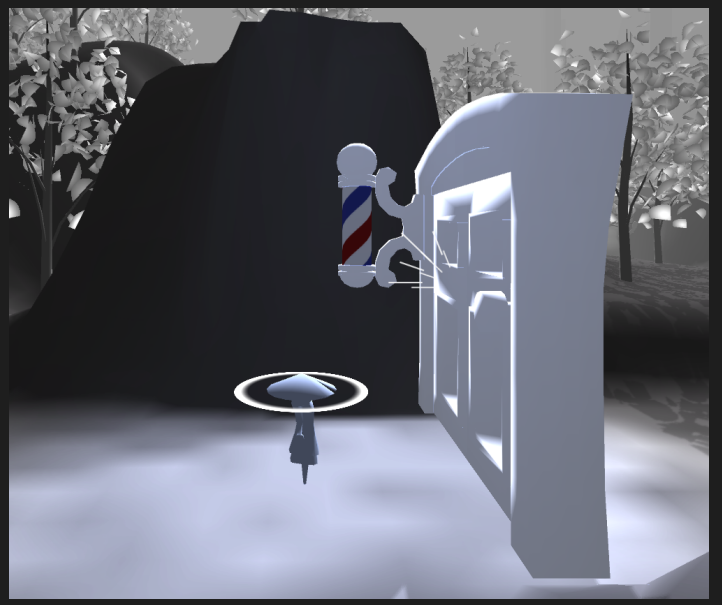


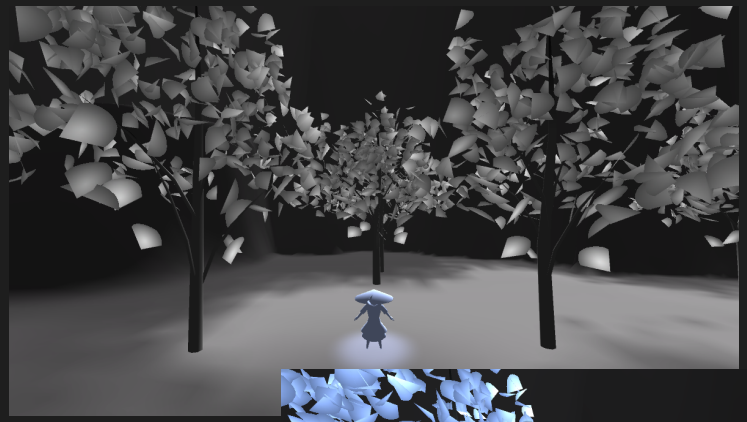
Trees shake and gain color when receiving a heartbeat.



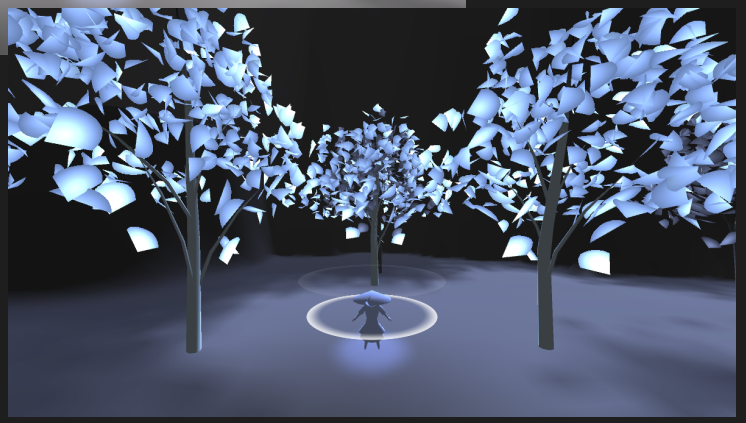
Different objects react to the heartbeat differently.

This door will open only when you time your heartbeat properly.

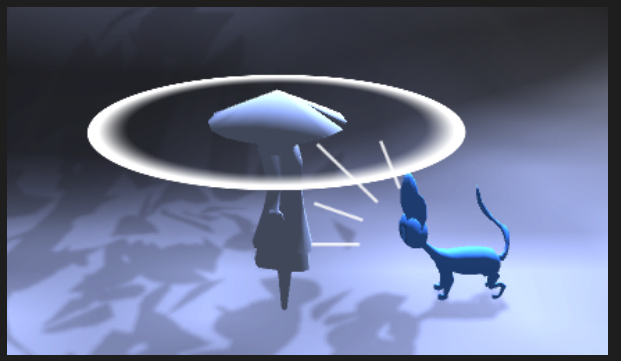




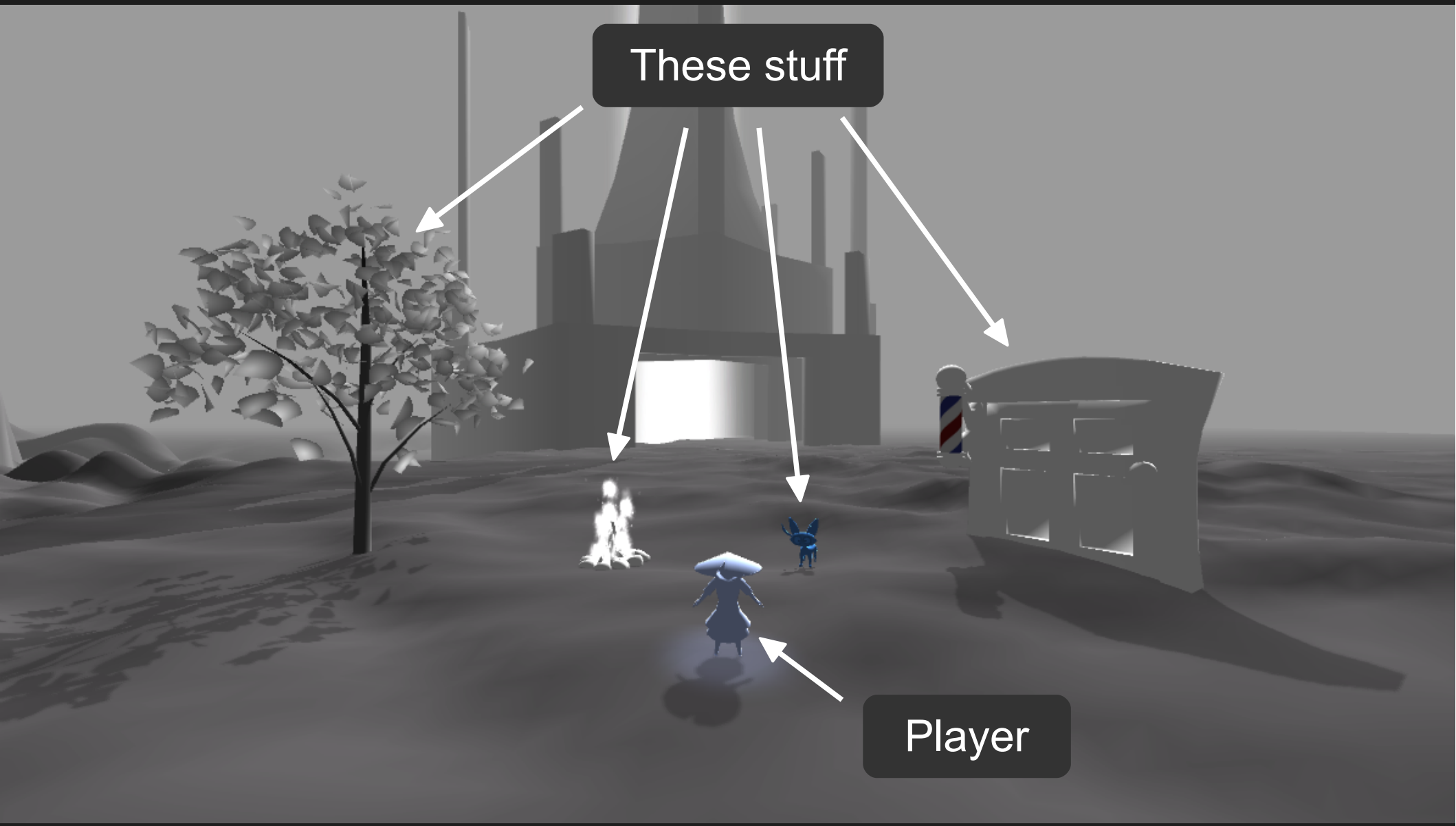
Trees shake and gain color when receiving a heartbeat.

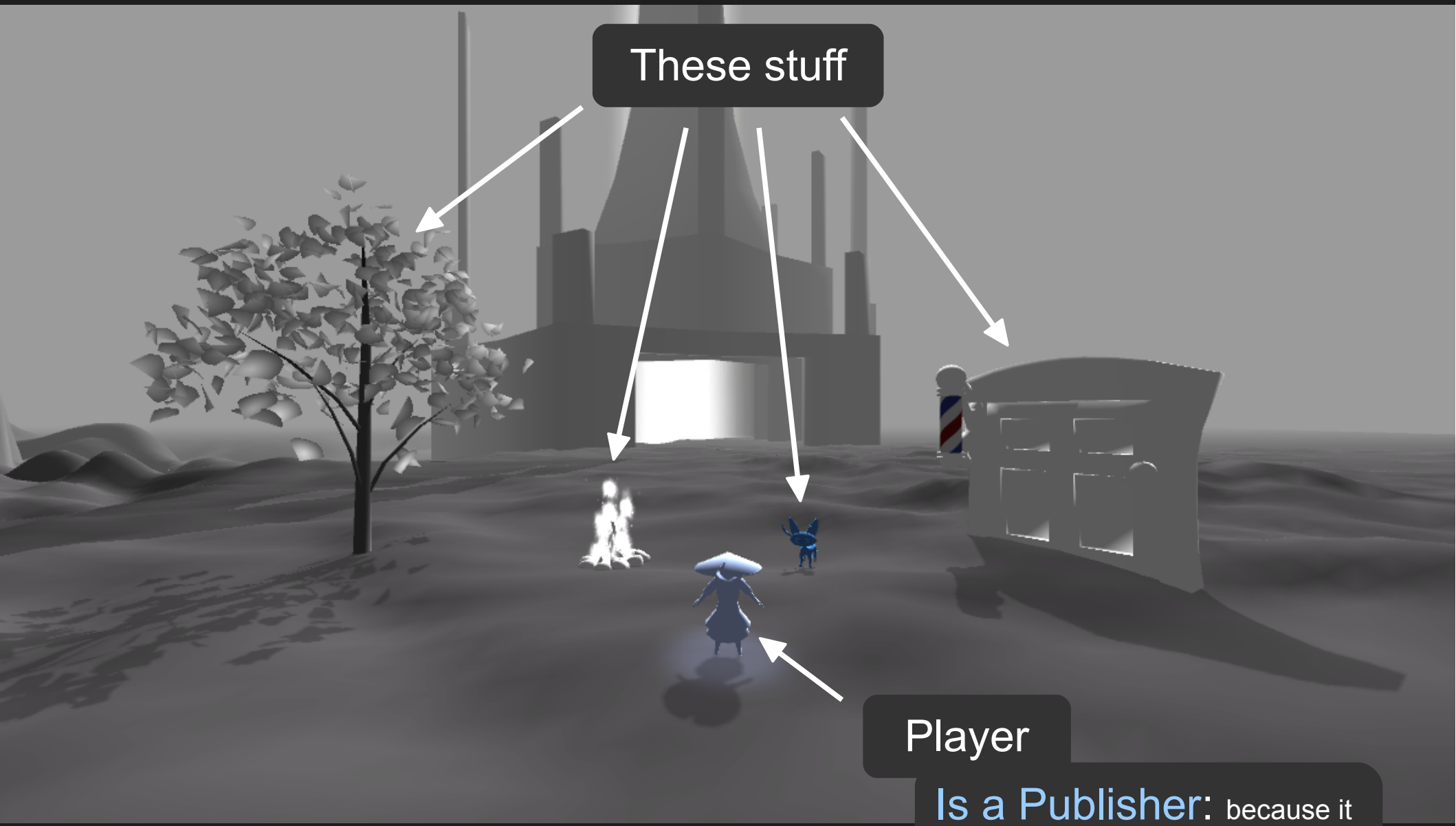


The cat will mimic your heartbeat timings and help you on your journey.




Player


These stuff
Player


These stuff
Player
Is a Publisher: because it broadcasts “heartbeat” messages to Subscribers (i.e. the stuff)


These stuff
Are Subscribers: because they listen for "heartbeat" messages from the Publisher (i.e. the Player)
Player
Is a Publisher: because it broadcasts "heartbeat" messages to Subscribers (i.e. the stuff)

So how does it look in code

Here's how the basic Subscriber type may look like:

```
public abstract class BeatListener : MonoBehaviour
{
}
```

So how does it look in code

Here's how the basic Subscriber type may look like:

```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}
```

We'll make a function that will be called when Subscribers receive beat messages.

So how does it look in code

Here's how the basic Subscriber type may look like:

```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}
```

We'll get back to this class later.

So how does it look in code

Here's how the Publisher may look like:

```
public class BeatBroadcaster : MonoBehaviour
{
}
```

So how does it look in code

Here's how the Publisher may look like:

```
public class BeatBroadcaster : MonoBehaviour
{
  List<BeatListener> _listeners = new List<BeatListener>();
}
```

First, we need a list to hold our Subscribers.

So how does it look in code

Here's how the Publisher may look like:

```
public class BeatBroadcaster : MonoBehaviour
{
  List<BeatListener> _listeners = new List<BeatListener>();

  public void RegisterListener(BeatListener b)
  {
    _listeners.Add(b);
  }
}
```

Then we need a function to add Subscribers to our list.

Note: There should actually be an UnregisterListener, but I did not put it in for brevity.

So how does it look in code

Here's how the Publisher may look like:

```
public class BeatBroadcaster : MonoBehaviour
{
  List<BeatListener> _listeners = new List<BeatListener>();

  public void RegisterListener(BeatListener b)
  {
    _listeners.Add(b);
  }

  void SignalBeatToListeners()
  {
    foreach (BeatListener b in _listeners)
    {
      b.OnBeat(transform.position);
    }
  }
}
```

Then we need a function to call the OnBeat message of all Subscribers.

So how does it look in code

Here's how the Publisher may look like:

```
public class BeatBroadcaster : MonoBehaviour
{
  List<BeatListener> _listeners = new List<BeatListener>();

  public void RegisterListener(BeatListener b)
  {
    _listeners.Add(b);
  }

  void SignalBeatToListeners()
  {
    foreach (BeatListener b in _listeners)
    {
      b.OnBeat(transform.position);
    }
  }

  void Update()
  {
    if (Input.GetButtonDown(“Fire1”))
    {
      SignalBeatToListeners();
    }
  }
}
```

For this sample, when we click the mouse, that's when we send our heartbeat message.

So how does it look in code

Here's how the Publisher may look like:

```
public class BeatBroadcaster : MonoBehaviour
{
  List<BeatListener> _listeners = new List<BeatListener>();

  public void RegisterListener(BeatListener b)
  {
    _listeners.Add(b);
  }

  void SignalBeatToListeners()
  {
    foreach (BeatListener b in _listeners)
    {
      b.OnBeat(transform.position);
    }
  }

  void Update()
  {
    if (Input.GetButtonDown("Fire1"))
    {
      SignalBeatToListeners();
    }
  }

  public static void RegisterListenerToPlayer(BeatListener b)
  {
    GameObject playerObj = GameObject.FindWithTag("Player");
    playerObj.GetComponent<BeatBroadcaster>().RegisterListener(b);
  }
}
```

And here's a convenience function to register a subscriber to the player game object.

We'll assume the player already has a BeatBroadcaster in it.

So how does it look in code

Here's how the basic Subscriber type may look like:

```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}
```

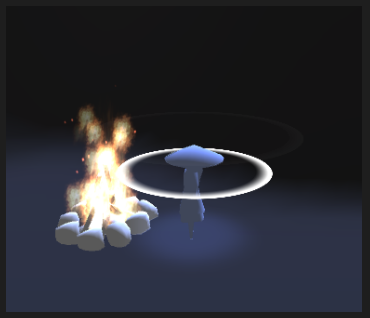
To actually create our subscribers, we need to subclass from our BeatListener class.

So how does it look in code

Here's how the basic Subscriber type may look like:

```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}
```

```
public class ParticleBeatListener : BeatListener
{
}
```

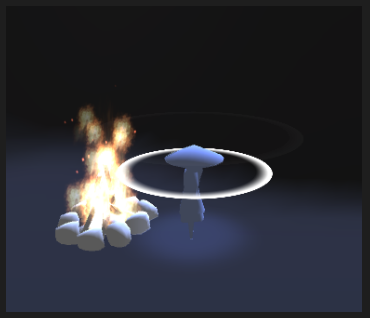


This will be one such subclass: the ParticleBeatListener. It will make particles move when it receives a beat message.

So how does it look in code

Here's how the basic Subscriber type may look like:

```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}

public class ParticleBeatListener : BeatListener
{
  public override void OnBeat(Vector3 beatSourcePosition)
  {
    // let particles play at normal speed
  }
}
```

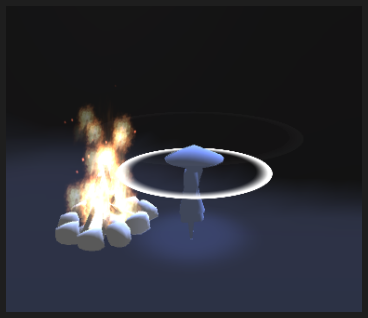


We'll override the OnBeat message, so we'll know when we get the beat message.

So how does it look in code

Here's how the basic Subscriber type may look like:

```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}

public class ParticleBeatListener : BeatListener
{
  ParticleSystem _particles;

  public override void OnBeat(Vector3 beatSourcePosition)
  {
    // let particles play at normal speed
  }
}
```

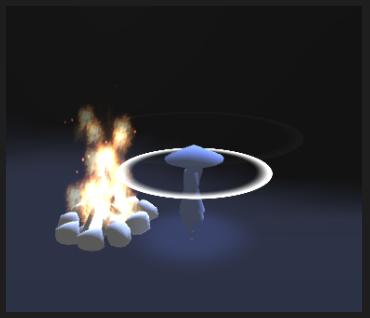


To add the code that will actually do the trick, we need to get ahold of our ParticleSystem component in a variable.

So how does it look in code

Here's how the basic Subscriber type may look like:

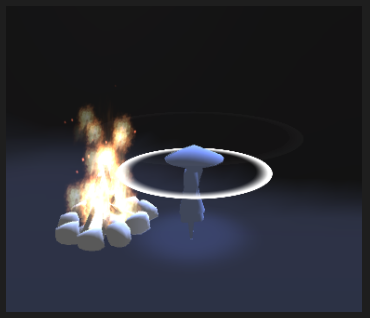
```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}

public class ParticleBeatListener : BeatListener
{
  ParticleSystem _particles;

  public override void OnBeat(Vector3 beatSourcePosition)
  {
    // let particles play at normal speed
    _particles.playbackSpeed = 1.0f;
  }
}
```



And here's the actual code to make the particles play.

So how does it look in code

Here's how the basic Subscriber type may look like:

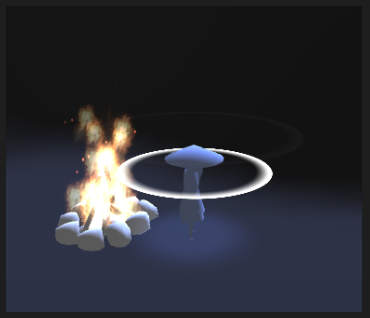


```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}

public class ParticleBeatListener : BeatListener
{
  ParticleSystem _particles;

  void Start()
  {
    _particles = GetComponent<ParticleSystem>();
  }

  public override void OnBeat(Vector3 beatSourcePosition)
  {
    // let particles play at normal speed
    _particles.playbackSpeed = 1.0f;
  }
}
```

But we need code to initialize our ParticleSystem variable.

So how does it look in code

Here's how the basic Subscriber type may look like:

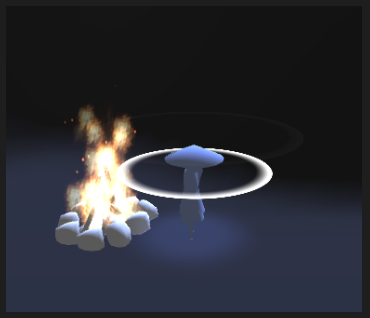


```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}

public class ParticleBeatListener : BeatListener
{
  ParticleSystem _particles;

  void Start()
  {
    _particles = GetComponent<ParticleSystem>();
  }

  public override void OnBeat(Vector3 beatSourcePosition)
  {
    // let particles play at normal speed
    _particles.playbackSpeed = 1.0f;
  }

  void Update()
  {
    // let particles slow down
    _particles.playbackSpeed -= 0.8f * Time.deltaTime;
  }
}
```

We also need code to let the particles slow down to a halt.

So how does it look in code

Here's how the basic Subscriber type may look like:

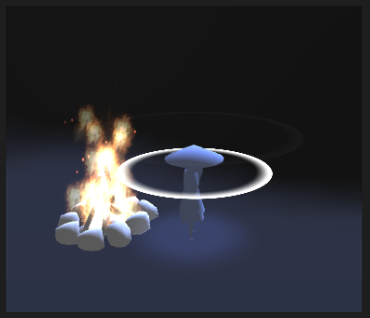


```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}

public class ParticleBeatListener : BeatListener
{
  ParticleSystem _particles;

  void Start()
  {
    _particles = GetComponent<ParticleSystem>();
    _particles.playbackSpeed = 0.0f;
  }

  public override void OnBeat(Vector3 beatSourcePosition)
  {
    // let particles play at normal speed
    _particles.playbackSpeed = 1.0f;
  }

  void Update()
  {
    // let particles slow down
    _particles.playbackSpeed -= 0.8f * Time.deltaTime;
  }
}
```

We also need to make sure particles are paused at the start.

So how does it look in code

Here's how the basic Subscriber type may look like:



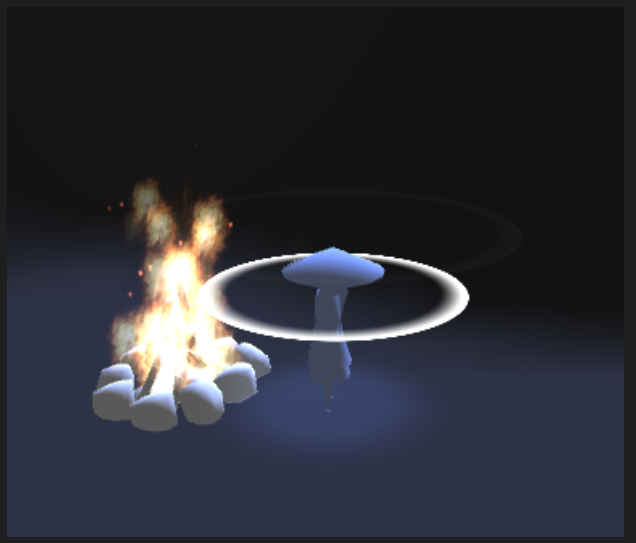
```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}

public class ParticleBeatListener : BeatListener
{
  ParticleSystem _particles;

  void Start()
  {
    BeatBroadcaster.RegisterListenerToPlayer(this);
    _particles = GetComponent<ParticleSystem>();
    _particles.playbackSpeed = 0.0f;
  }

  public override void OnBeat(Vector3 beatSourcePosition)
  {
    // let particles play at normal speed
    _particles.playbackSpeed = 1.0f;
  }

  void Update()
  {
    // let particles slow down
    _particles.playbackSpeed -= 0.8f * Time.deltaTime;
  }
}
```

Finally, the most crucial thing is to let our class receive beat messages. We make use of our convenience function here.

So how does it look in code

Here's how the basic Subscriber type may look like:

```
public abstract class BeatListener : MonoBehaviour
{
  // override to get beat messages
  public abstract void OnBeat(Vector3 beatSourcePosition);
}

public class ParticleBeatListener : BeatListener
{
  ParticleSystem _particles;

  void Start()
  {
    BeatBroadcaster.RegisterListenerToPlayer(this);
    _particles = GetComponent<ParticleSystem>();
    _particles.playbackSpeed = 0.0f;
  }

  public override void OnBeat(Vector3 beatSourcePosition)
  {
    // let particles play at normal speed
    _particles.playbackSpeed = 1.0f;
  }

  void Update()
  {
    // let particles slow down
    _particles.playbackSpeed -= 0.8f * Time.deltaTime;
  }
}
```



Why use Design Patterns?

Why use Design Patterns?

Turns out learning object-oriented programming is not enough!

Why use Design Patterns?

Turns out learning object-oriented programming is not enough!

Just knowing how to write classes doesn't mean you'll automatically make good ones.

Why use Design Patterns?

Turns out learning object-oriented programming is not enough!

Just knowing how to write classes doesn't mean you'll automatically make good ones.

You have to know how to design them better!

Why use Design Patterns?

Turns out learning object-oriented programming is not enough!

Just knowing how to write classes doesn't mean you'll automatically make good ones.

You have to know how to design them better!

Designs that make your code...

Why use Design Patterns?

Turns out learning object-oriented programming is not enough!

Just knowing how to write classes doesn't mean you'll automatically make good ones.

You have to know how to design them better!

Designs that make your code...

Easier to read!

Why use Design Patterns?

Turns out learning object-oriented programming is not enough!

Just knowing how to write classes doesn't mean you'll automatically make good ones.

You have to know how to design them better!

Designs that make your code...

Easier to read!

Easier to modify! Can cope with change!

Why use Design Patterns?

Turns out learning object-oriented programming is not enough!

Just knowing how to write classes doesn't mean you'll automatically make good ones.

You have to know how to design them better!

Designs that make your code...

Easier to read!

Easier to modify! Can cope with change!

Easier to maintain! Easy to fix errors!

Why use Design Patterns?

Design Patterns
can help with that goal!

Why use Design Patterns?

Reusable! You can reuse these designs in many projects.

Why use Design Patterns?

Reusable! You can reuse these designs in many projects.

...but Adaptable! Design patterns are not *actual* source code, they are only the *design* for your source code. You can use them for a myriad of situations.

For a video game!
Or a database application!
Or robot overlords!

Why use Design Patterns?

Reusable! You can reuse these designs in many projects.

...but Adaptable! Design patterns are not *actual* source code, they are only the *design* for your source code. You can use them for a myriad of situations.

For a video game!
Or a database application!
Or robot overlords!

Battle-tested! Many people refined these designs over time. You can be sure any errors in these designs have been corrected a long time ago.

Why use Design Patterns?

Reusable! You can reuse these designs in many projects.

...but Adaptable! Design patterns are not *actual* source code, they are only the *design* for your source code. You can use them for a myriad of situations.

For a video game!
Or a database application!
Or robot overlords!

Battle-tested! Many people refined these designs over time. You can be sure any errors in these designs have been corrected a long time ago.

Shared terminology! It's easier to communicate class designs to teammates if we have labels for them.

When not to use Design Patterns!

When not to use Design Patterns!

They make your code
more complex
though!

When not to use Design Patterns!

They make your code
more complex
though!

So ask yourself, “Do I *really* need to use a design pattern here?”

When not to use Design Patterns!

They make your code
more complex
though!

So ask yourself, “Do I *really* need to use a design pattern here?”

If you can use a simpler solution that doesn't rely on patterns, try it out!

When not to use Design Patterns!

They make your code
more complex
though!

So ask yourself, “Do I *really* need to use a design pattern here?”

If you can use a simpler solution that doesn't rely on patterns, try it out!

Remember, there are still times when you can make code that's easier to read, change, and maintain, without needing patterns!

When not to use Design Patterns!

They make your code
more complex
though!

So ask yourself, “Do I *really* need to use a design pattern here?”

If you can use a simpler solution that doesn't rely on patterns, try it out!

Remember, there are still times when you can make code that's easier to read, change, and maintain, without needing patterns!

Make sure you *really* need it before you use it!

When not to use Design Patterns!

They make your code
more complex
though!

So ask yourself, "Do I *really* need to use a design pattern here?"

If you can use a simpler solution that doesn't rely on patterns, try it out!

Remember, there are still times when you can make code that's easier to read, change, and maintain, without needing patterns!

Make sure you *really* need it before you use it!

"When you have a shiny new hammer, suddenly, everything starts looking like a nail."

- Bernard Baruch

When not to use Design Patterns!

They take
longer to
implement!

When not to use Design Patterns!

They take
longer to
implement!

If you're rushing for a deadline, maybe a brute-forced design will work better!

When not to use Design Patterns!

They take
longer to
implement!

If you're rushing for a deadline, maybe a brute-forced design will work better!

Just make sure to improve your code later!

When not to use Design Patterns!

They take
longer to
implement!

If you're rushing for a deadline, maybe a brute-forced design will work better!

Just make sure to improve your code later!

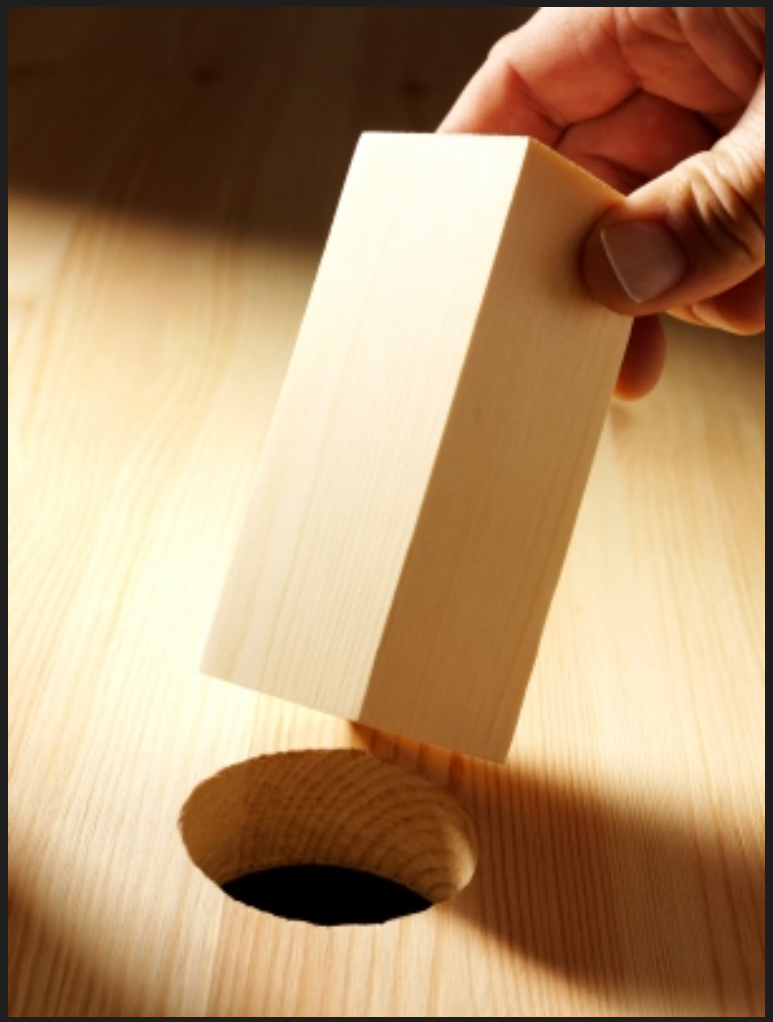
“A good enough plan that can be executed right now is better than a perfect plan that you're not sure when can be executed.”

-George Patton
(paraphrased)

In the end, always
weigh your
options!

In the end, always weigh your options!

Remember, right tool for the right job!



Remember!

Remember!

Patterns are not rules to blindly follow!

Remember!

Patterns are not rules to blindly follow!

Patterns are just another set of tools in your toolbox!

Remember!

Patterns are not rules to blindly follow!

Patterns are just another set of tools in your toolbox!

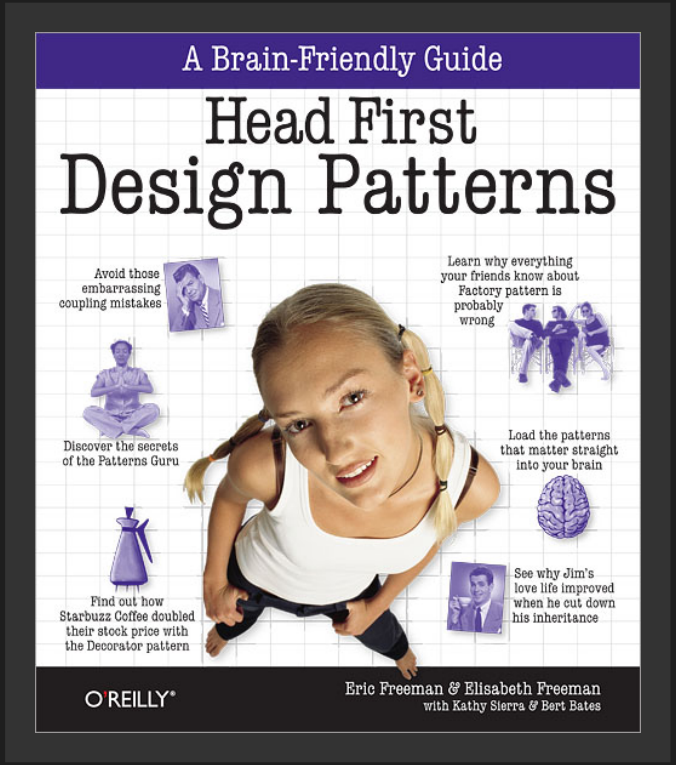
Don't be afraid to tweak and modify them if it solves your problem better!

Recommended books

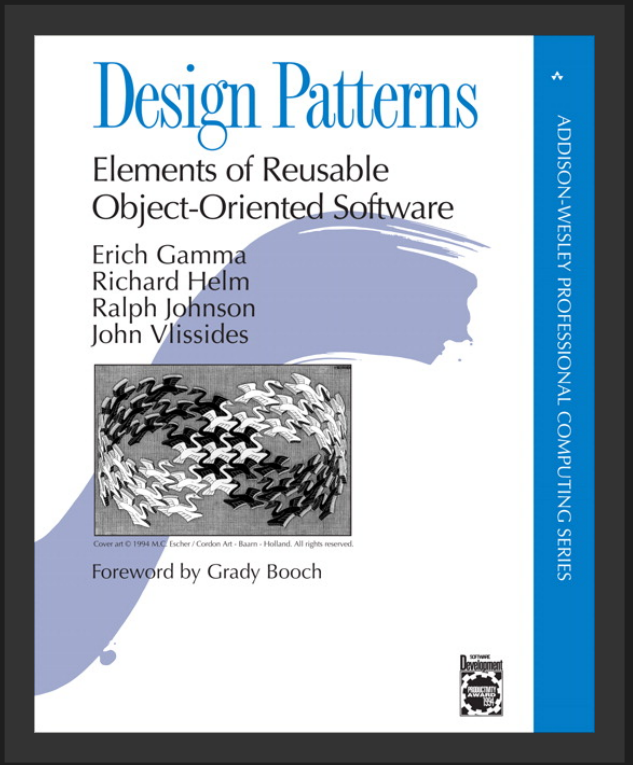
Take note:

You better have a fairly good grasp of your programming language of choice (an object-oriented one) before you start learning this.

(click on the book cover for a link where to buy it)



Head First Design Patterns



Design Patterns

Addendum

Wait a minute, your code feels like a roundabout way just to play and pause particles!

> *“All you're doing is gathering a bunch of classes and calling their OnBeat function whenever the mouse is clicked. Why bother with that design?*
>
> *Couldn't you have just used a trigger sphere or Physics.OverlapSphere to collect surrounding BeatListeners to achieve the same thing?”*

Yes, that's true. But having a Publisher-Subscriber relationship is more flexible (though admittedly at the cost of more complexity).

What if I want to prioritize whose OnBeat gets called first? (i.e. it's easy to sort the list of subscribers and you'd need to sort it only once during start of the game, in this way).

What if mouse clicking isn't the only thing that sends beat messages? What if there are other sources of messages? What if I want something more crazy like an object who's a Subscriber *and also* a Publisher? Its easy to try that out if I have a Publisher-Subscriber code structure in the first place.

And I indeed anticipated that I want to experiment with such ideas beforehand. I wanted my code to be flexible. That is why I went with the Observer pattern.

But remember, more flexibility = more complexity.

A more flexible structure is nice because it encourages experimentation in game design, but it is also more complex. Flaws or bugs introduced in more complex code are harder to track down than in simpler code. (To be fair, that difficulty is reduced if you have lots of coding experience, so it's not as bad as you might think.)

The reverse also matters. Solutions that are simpler and more straightforward make it difficult to experiment ideas with, but flaws that occur in such code are also simpler, and thus easier to diagnose.

So, what should you choose? The answer is “whichever is better for your current situation”. It's a tradeoff you always have to be mindful of.

Since more flexible code is harder to debug, it makes sense to add flexibility only in places where it is really needed.

Take note, the source code I shared isn't the only way to implement the Observer Pattern.

For example, some people design it such that even the message that gets distributed is also a class.

Also take note that the Observer Pattern isn't the only way to achieve a Publisher-Subscriber relationship.

There are other styles, like the so-called Event Bus.

If the simple Observer Pattern is a one-way street, then the Event Bus is like a highway. (Perhaps a later presentation can explain this in more detail)

That is, some design patterns are more flexible (hence more complex) than others.

Sometimes you'll find more than one design pattern to be a valid solution to your problem. You'll have to weigh each one and choose what you think is the best approach. (It can also be a valid choice to not use any design pattern at all.)

Question me!

Ferdinand Joseph Fernandez

Chief Technological Officer, Dreamlords Digital, Inc.
Admin & Co-founder, Unity Philippines Users Group

 @AnomalusUndrdog

 victisgame.wordpress.com

 anomalousunderdog.blogspot.com