Dependency Injection
Plugin Development

Brian LaMarche

4-24-2012

Introduction

- We've spent the majority of class talking about
  - Coupling/Cohesion
  - Interface design
  - Users
  - Extensibility
- It really bounces back to a fundamental concept we were taught early on:
  - Make your code modular.

Modularity

- So we started off breaking our *int main(void)* into multiple functions that served a purpose
- This was very modular we told ourselves! Way to go team!
- But separate functions does not imply modularity, in fact we really have no idea what this really means
  - Just because you can make multiple functions does not mean you've reduced coupling

Coupling

- Coupling, as we have discussed, is a measure of interdependence of modules. It goes beyond functions/methods.
- In OOP, we talk about objects that are coupled together
- We know we should reduce this if we want these properties:
  - Extensibility – ability to add to our application easily
  - Maintainability – ability to fix bugs / modules easily

Frameworks and Packages

- We've briefly talked about different design patterns to help alleviate coupling.
- When we design we should think beyond the scope of an assembly, dll, library, etc.
- We have to consider the full package from user to developer.
  - If I gave this software to someone else, could they add to it without the need to compile the application?!
- Packages/Frameworks act as a basis for this.

Plugins

- Plugin development is one way to open our framework/application up to external developers.
- There are other motivation(s):
  - Deployment –
    - Don't recompile, re-install the application.
    - Create new plug-ins
  - Mock Objects for testing and offline
    - I don't have hardware right now.
    - test driven development
  - Licenses
    - Sometimes you want to provide a tool, but not provide your code you have a license for.

Plugins

- Plugins are those components that “plug-in” to the framework/application provided.
- Plug-ins are run-time configurable
- Plug-ins are dynamically, not statically linked.
    - At run-time something is in charge of creating them.

Plugins

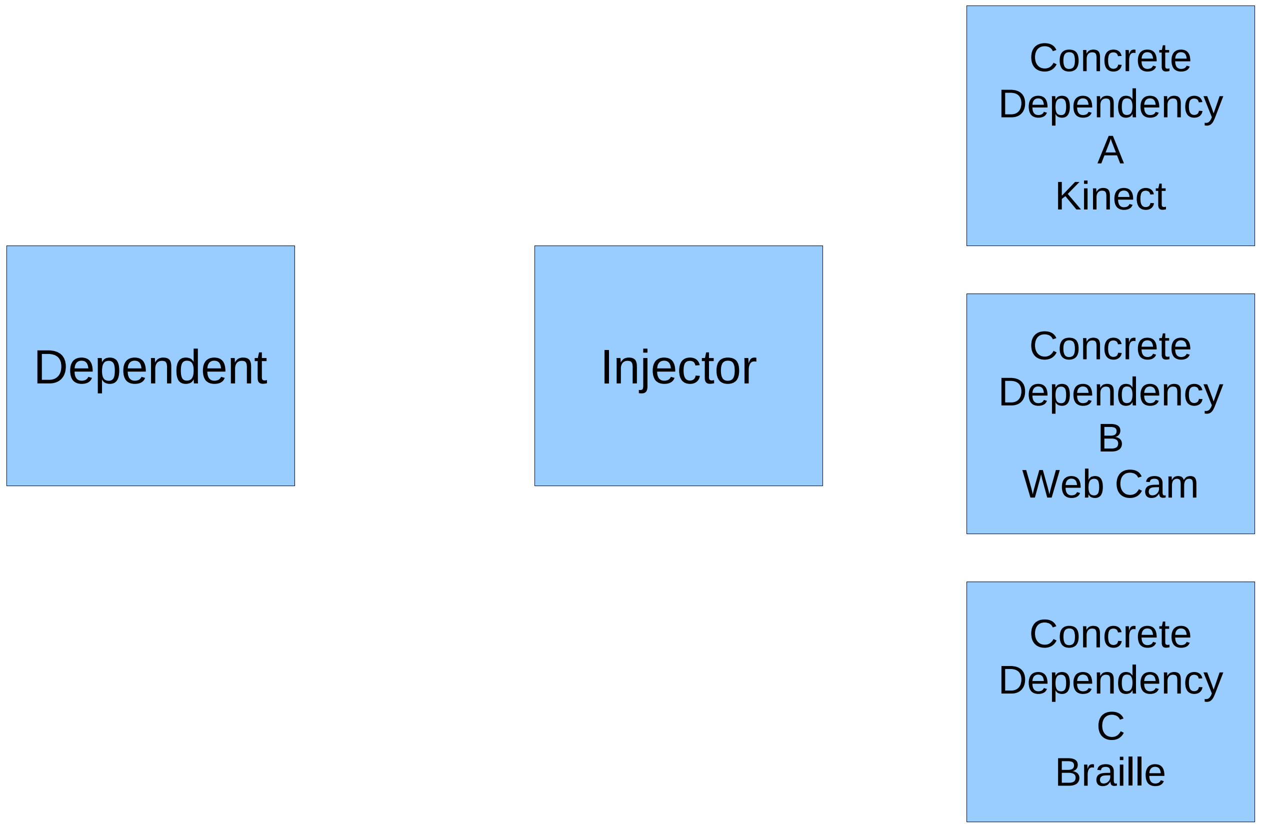
- Developing an application to use plug-ins means to design against an interface
  - You don't know anything about the concrete class.
  - You just know what interface it provides.

Dependency Injection

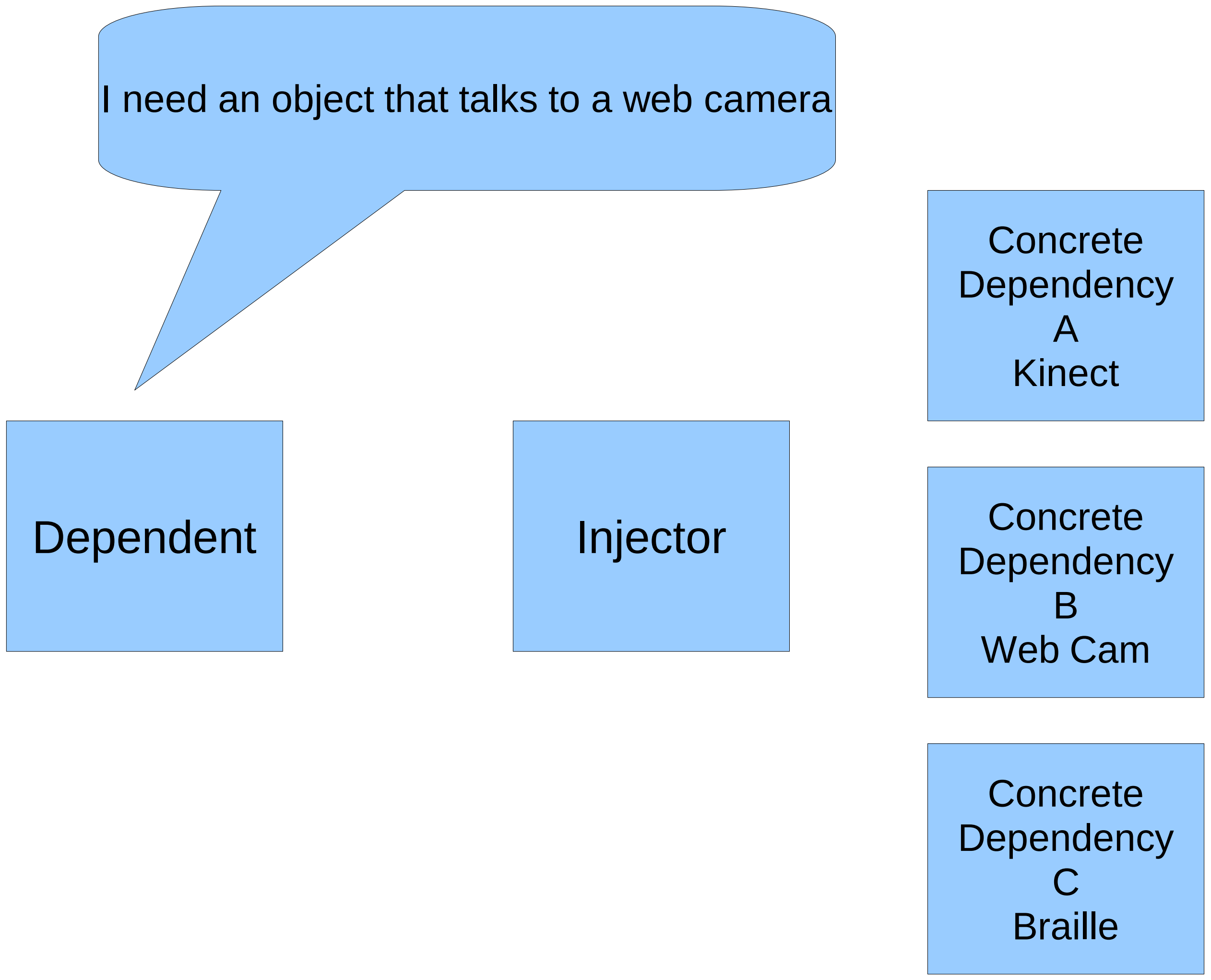
- Plugin design patterns are best described by dependency injection

- Allows a way for an object to be created at run time rather than at compile time.
- Contains:
  - Dependent – system that consumes
  - Injector – System that creates the objects
  - Dependencies – interfaces (plug-ins)

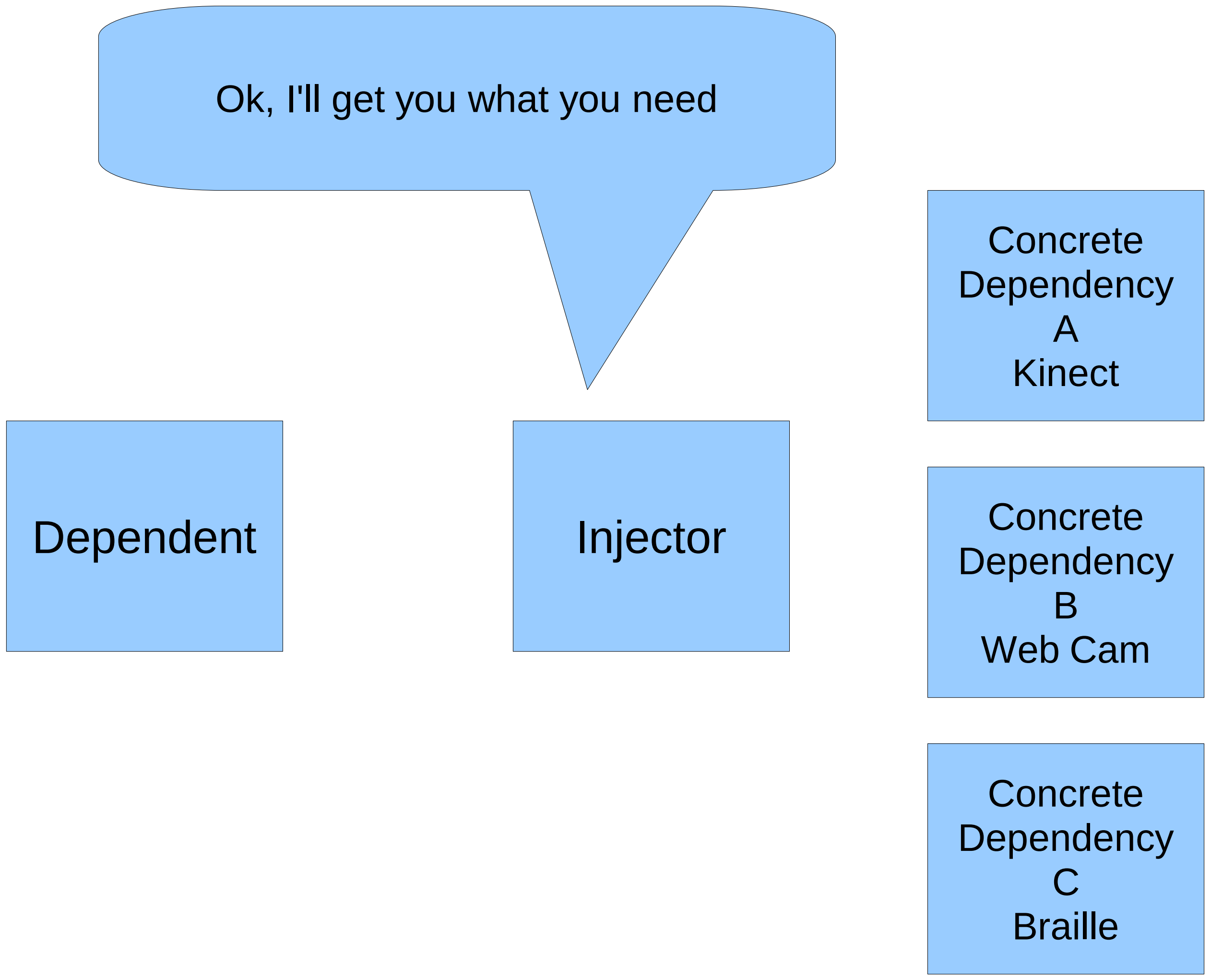
Dependency Injection


Dependent
Injector
Concrete Dependency A Kinect
Concrete Dependency B Web Cam
Concrete Dependency C Braille

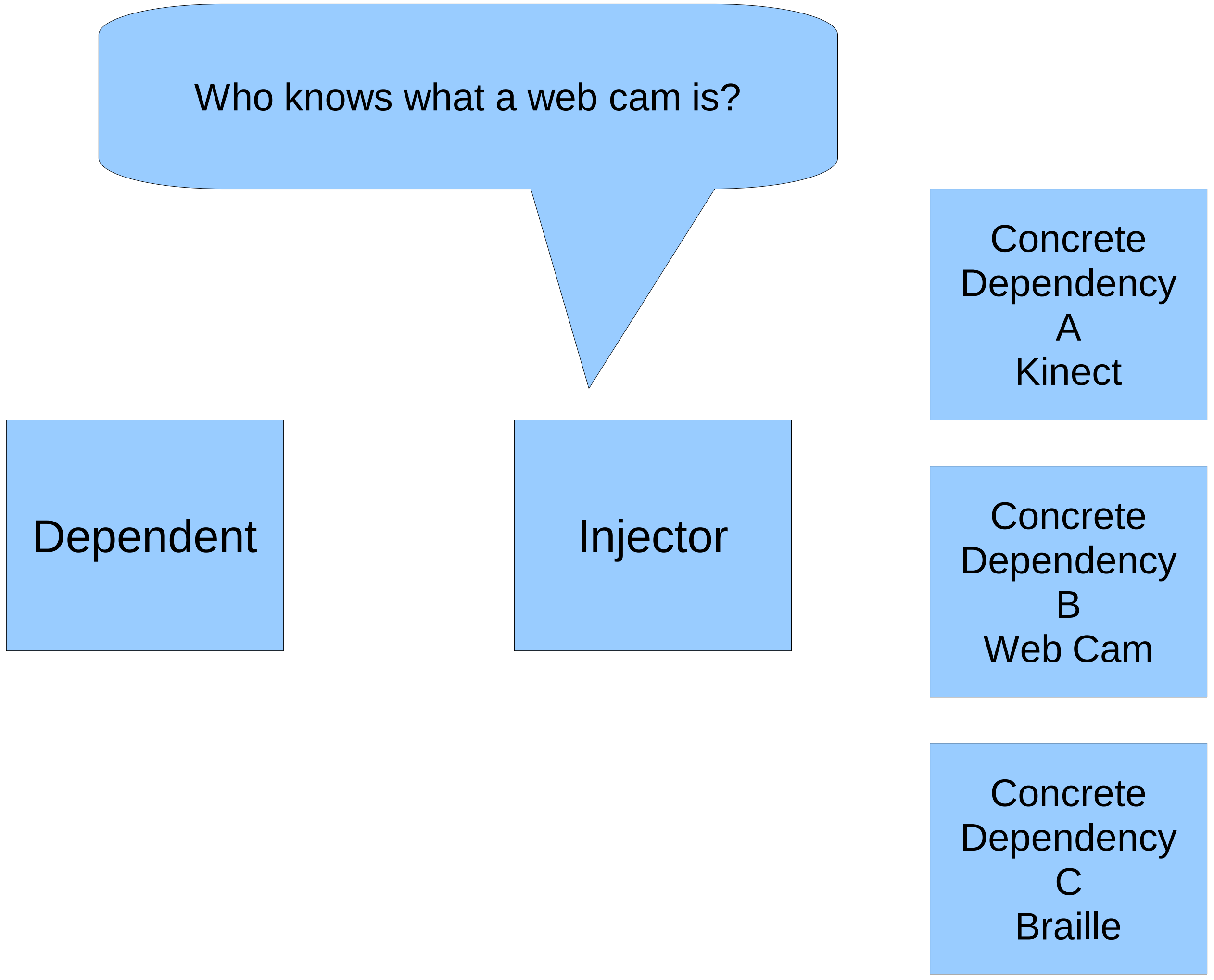
Dependency Injection


I need an object that talks to a web camera
Dependent
Injector
Concrete Dependency A Kinect
Concrete Dependency B Web Cam
Concrete Dependency C Braille

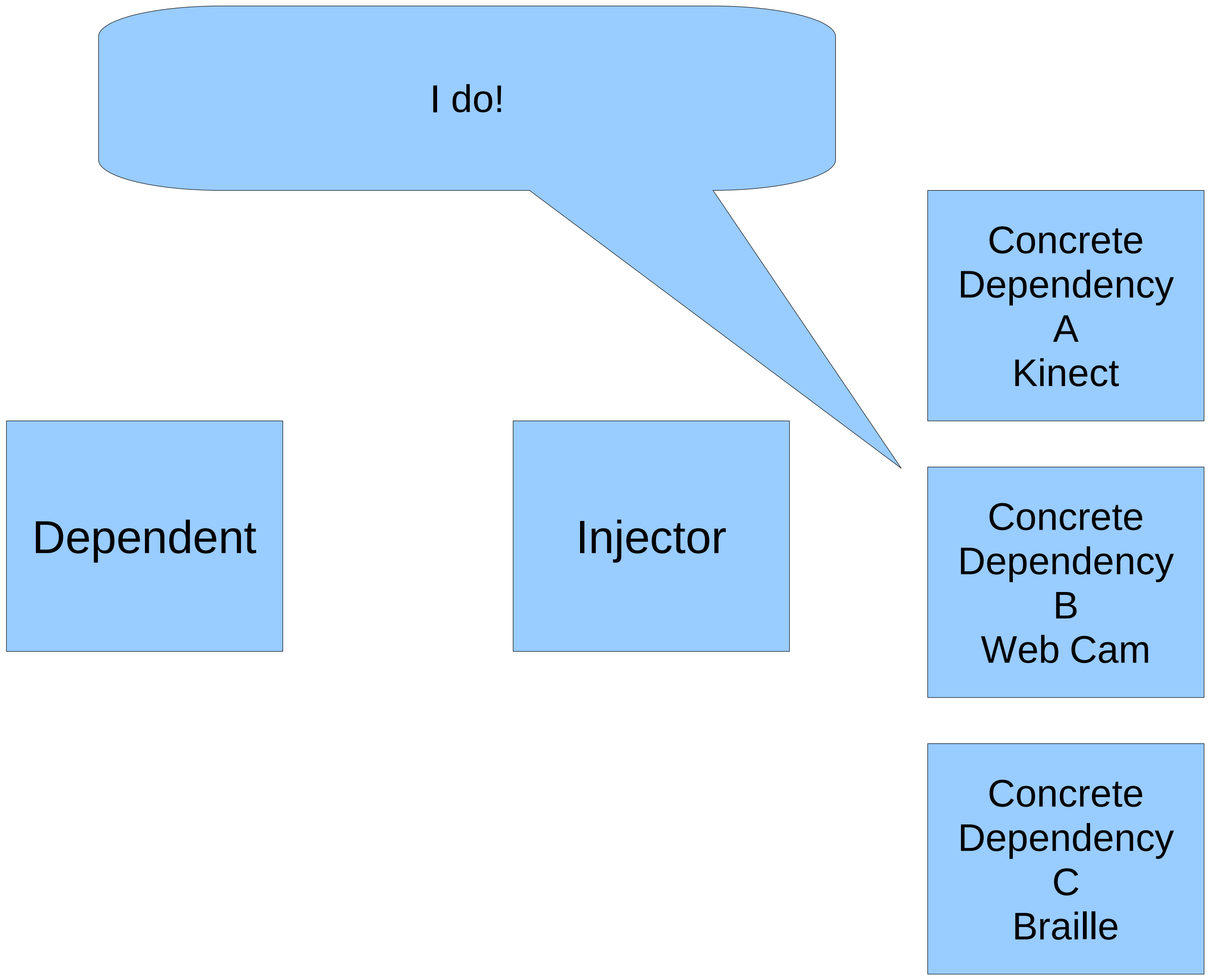
Dependency Injection


Ok, I'll get you what you need
Dependent
Injector
Concrete Dependency A Kinect
Concrete Dependency B Web Cam
Concrete Dependency C Braille

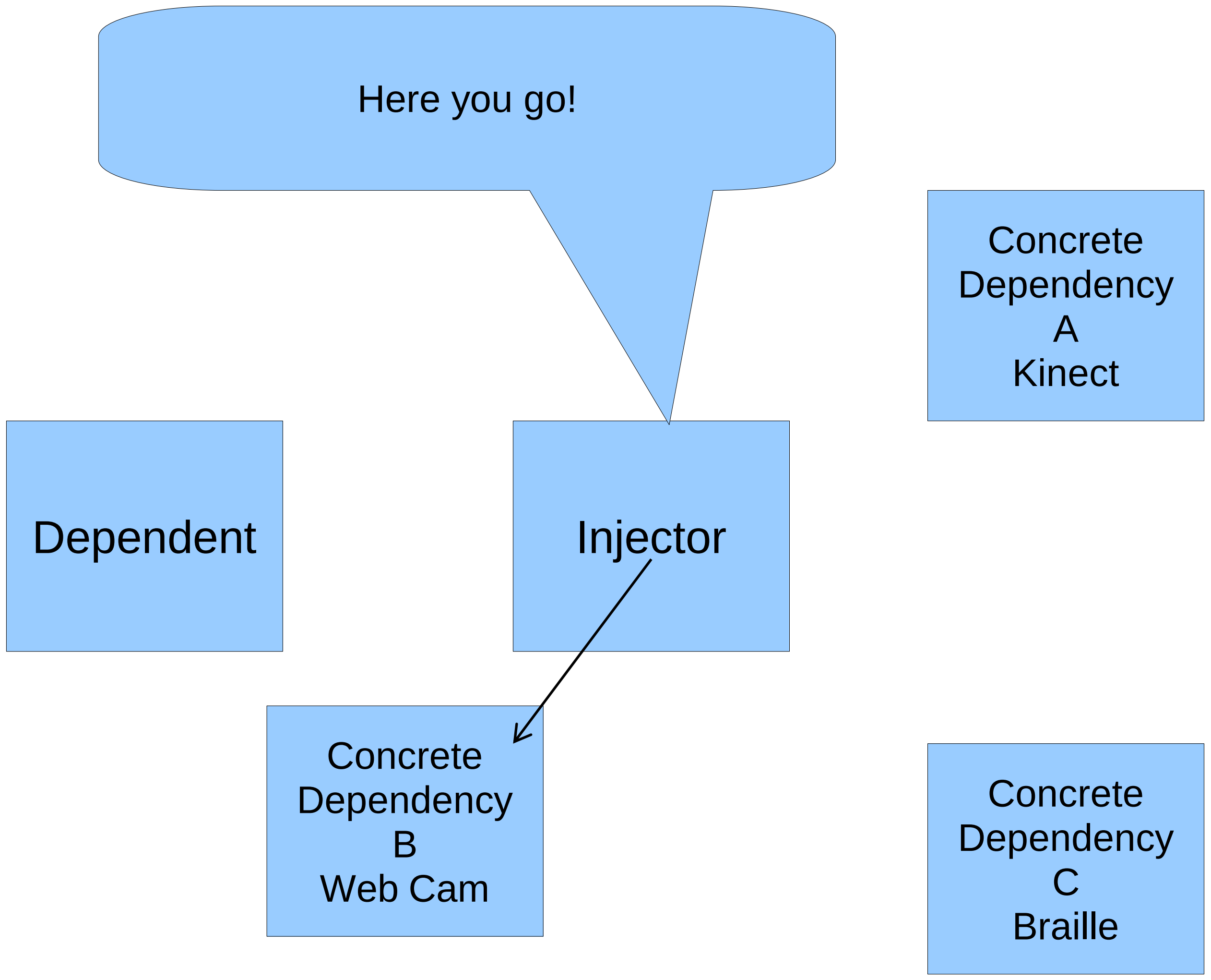
Dependency Injection


Who knows what a web cam is?
Dependent
Injector
Concrete Dependency A Kinect
Concrete Dependency B Web Cam
Concrete Dependency C Braille

Dependency Injection


I do!
Dependent
Injector
Concrete Dependency A Kinect
Concrete Dependency B Web Cam
Concrete Dependency C Braille

Dependency Injection


Here you go!
Concrete Dependency A Kinect
Dependent
Injector
Concrete Dependency B Web Cam
Concrete Dependency C Braille

Dependency Injection

- The previous example poses some questions:
  - How does the injector know what to instantiate?
  - How does the Dependent know what to request?

Dependency Injection

- The previous example poses some questions:
  - How does the injector know what to instantiate?
    - By design, this can be tricky. Usually you tag your dependency's with some kind of unique ID:
      - GUID
      - Hash
      - Name
      - Type

Dependency Injection

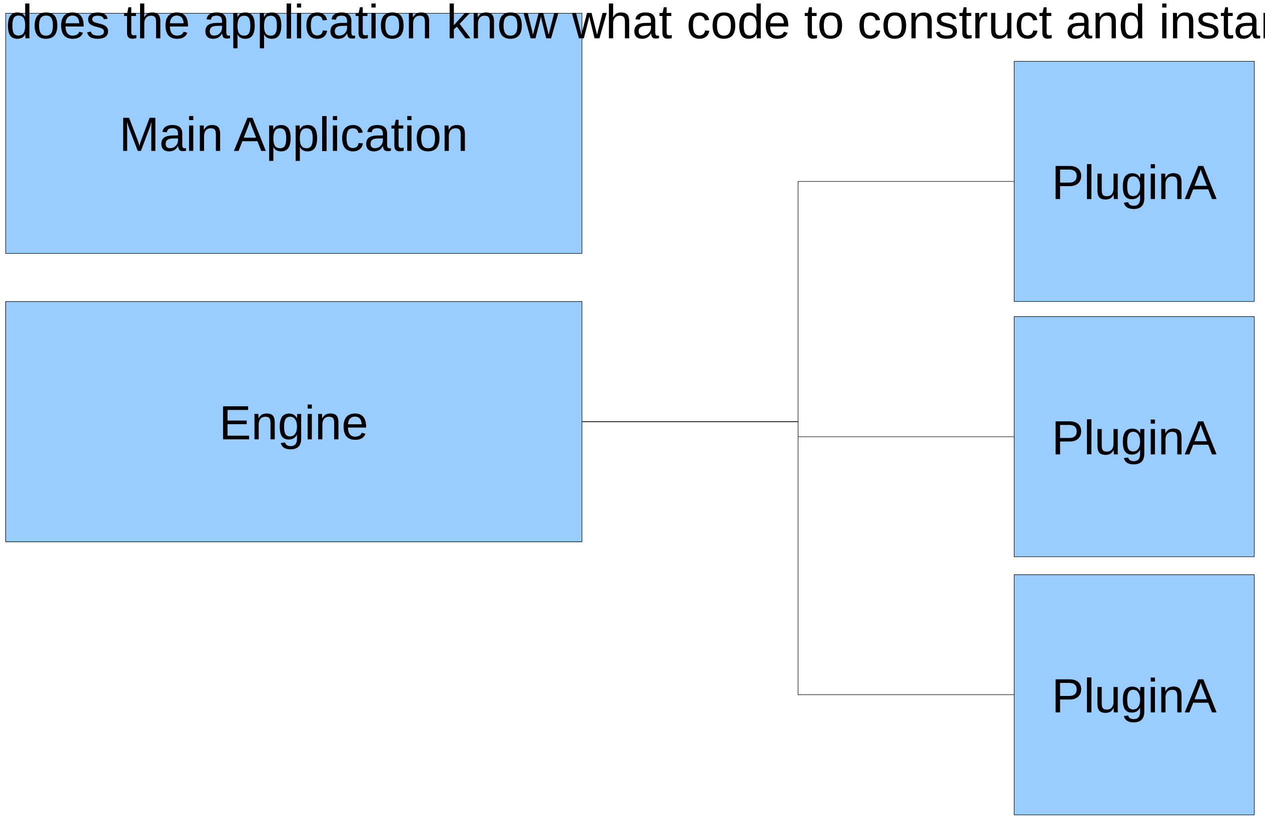
- The previous example poses some questions:
  - How does the injector know what to instantiate?
    - By design, this can be tricky. Usually you tag your dependency's with some kind of unique ID as meta-data:
      - GUID
      - Hash
      - Name
      - Type

Plugin

Let's recap:

A plugin is essentially a bit of code that extends the application. For .net it's an assembly,
a dll. Plugins are loaded at run-time and are not packaged as part of the application.

So how does the application know what code to construct and instantiate?

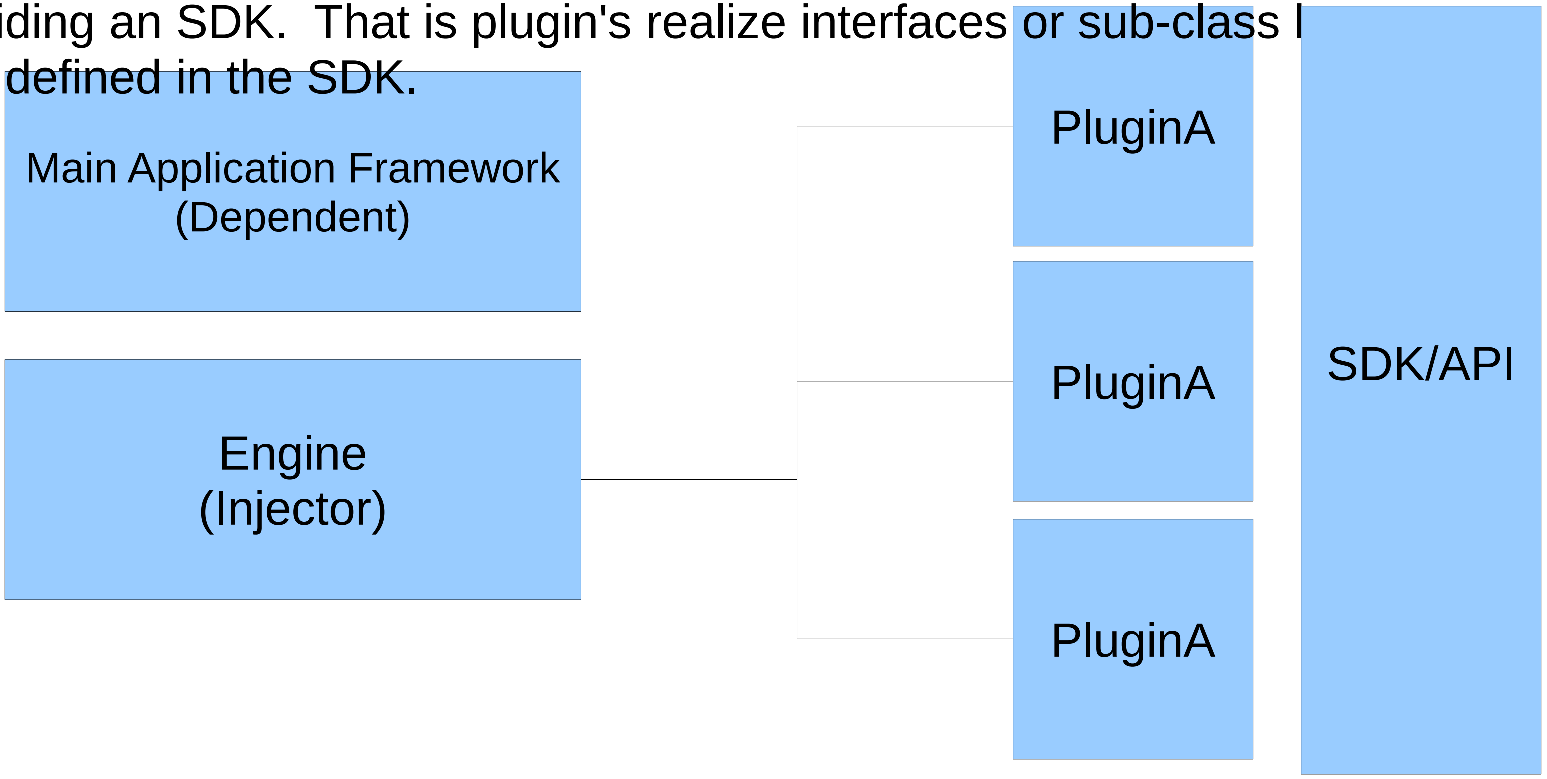


Plugin

Let's recap:

A plugin is essentially a bit of code that extends the application. For .net it's an assembly,
a dll. Plugins are loaded at run-time and are not packaged as part of the application.

So how does the application know what code to construct and instantiate? This is done
By providing an SDK. That is plugin's realize interfaces or sub-class l
classes defined in the SDK.

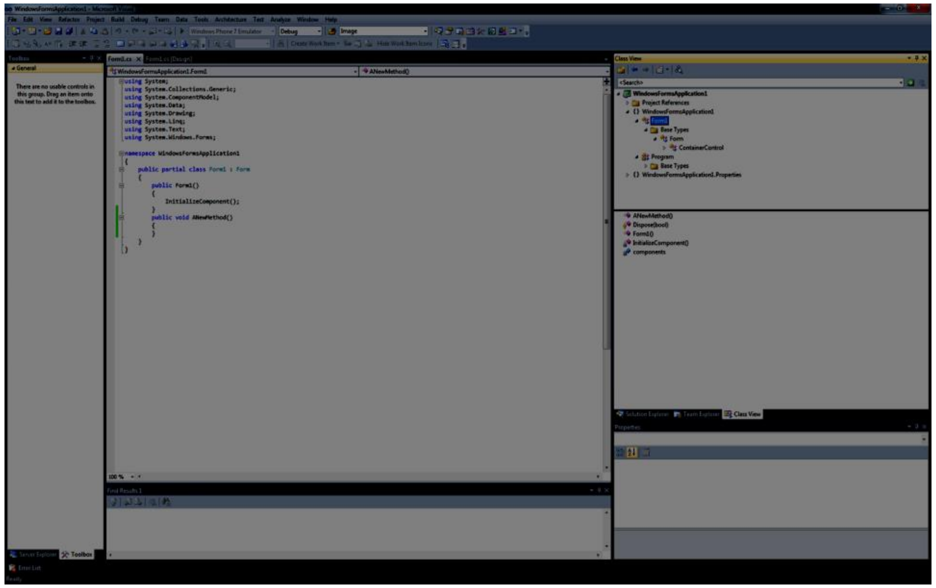


Meta-Data

- Meta-Data allows you to describe a concrete object so that the injector can find what it needs, when it needs it.
- The injector needs a way to “query” this information from a DLL at run-time
- The .net framework has System.Reflection
- System.Reflection is a namespace containing objects that allow us to interrogate a DLL.
  - Other languages, such as Python, have similar mechanisms.

Meta-Data

- Basically, given any DLL, I can look into the DLL and find out all the types that are available.
- Look at the Class View in Visual Studio. How do you think it can read a DLL that you wrote and tell you what types are available?



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        public void ANewMethod()
        {
        }
    }
}
```

Class View:

```
WindowsFormsApplication1
  Project References
  {} WindowsFormsApplication1
    Form1
      Base Types
        Form
          ContainerControl
    Program
      Base Types
  {} WindowsFormsApplication1.Properties
```

```
ANewMethod()
Dispose(bool)
Form1()
InitializeComponent()
components
```

Meta-Data

- How do Types associate with Meta-Data? From MSDN:
  - Type is the root of the System.Reflection functionality and is the primary way to access metadata. Use the members of Type to get information about a type declaration, such as the constructors, methods, fields, properties, and events of a class, as well as the module and the assembly in which the class is deployed.

Meta-Data

- So if we know an objects type, we can say something about how to create it.
- Moreover, we can look at it's interface (Properties/Methods/Attributes if publicly exposed!)
- This means, for a given DLL, I can see every publicly exposed type (object) that I can instantiate!

Meta-Data

- That means I just need to know what kind of types to create?!?
- An injector has to ask for a specific type like a WebCam
  - But how do you program for that?

```
WebCam w = Injector.CreateWebCam("logitech");
```

  - What if you want to use a kinect?

```
Kinect w = Injector.CreateWebCam("microsoft");
```

  - Well, if those objects are defined in a library loaded at run-time, the above code will not work!

Meta-Data

- So we use interfaces (think strategy!!!!)

```
IVision w = Injector.CreateWebCam("logitech");
```

- Ok, so if we can create objects that realize the given interface, we can create objects at run-time using meta-data
    - (in this example we are using a string to denote what type)
- NOTE: There are ways to take a type and figure out if it realizes a specific interface. We'll talk about this more later. For now assume that this is possible.

Also...

- Notice how this example looks a lot like a factory?
- It is, when you implement it. The injector basically is a factory, that knows how to extract information from the concrete dependencies.
- This pattern plays off of strategy as well as you can tell.

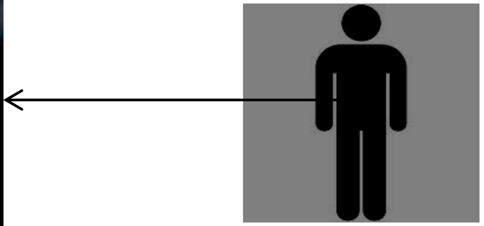
Putting this together

- Ok, so we want to allow our application to be extended
- We apply the Dependency Injection to do this
- We use .net Reflection to help us query object meta-data
- We know that if we use this technology and pattern, we can create objects at run-time, without having to know anything about their concrete implementation
- BUT! – we have to get the interface right!

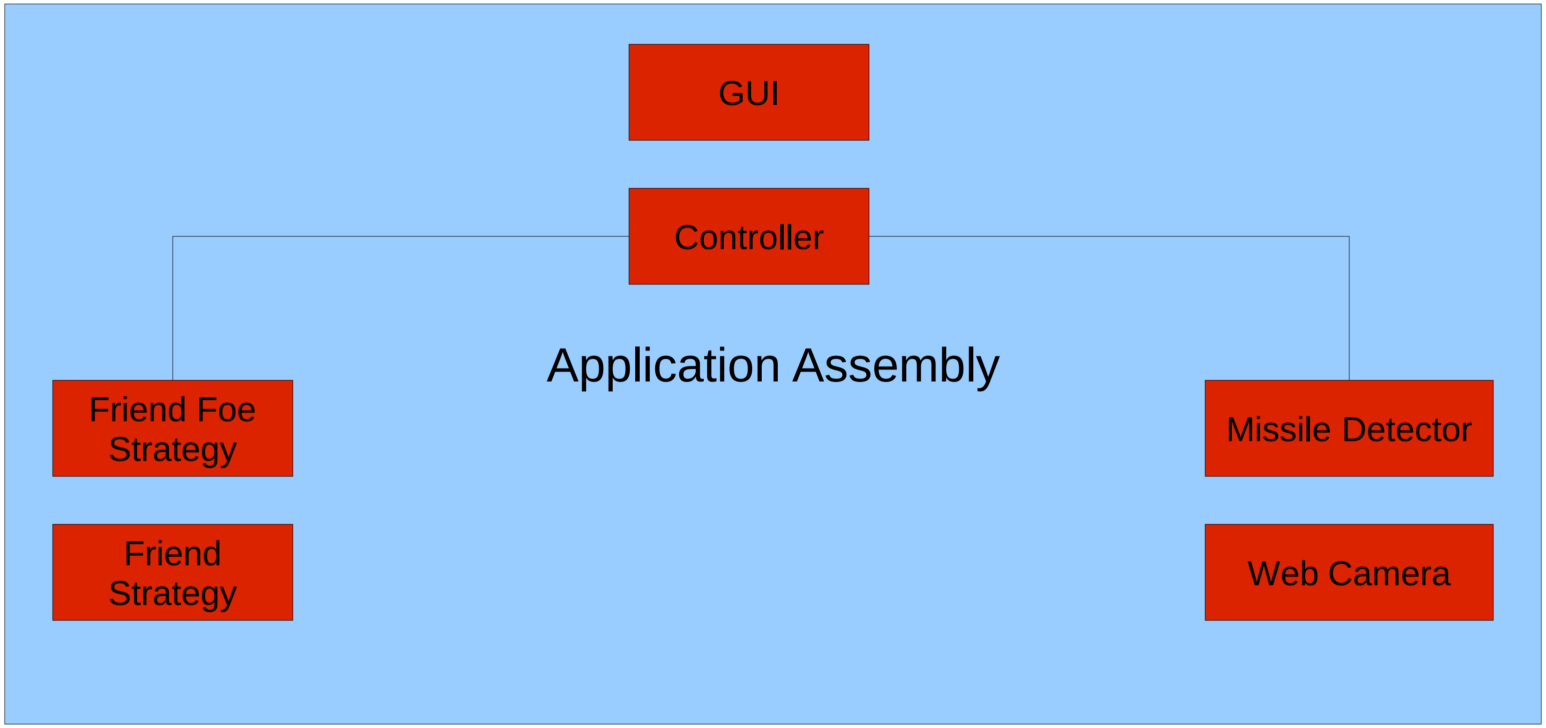
So let's stop for a second and put this in perspective of the project....

Project System

Our Target System



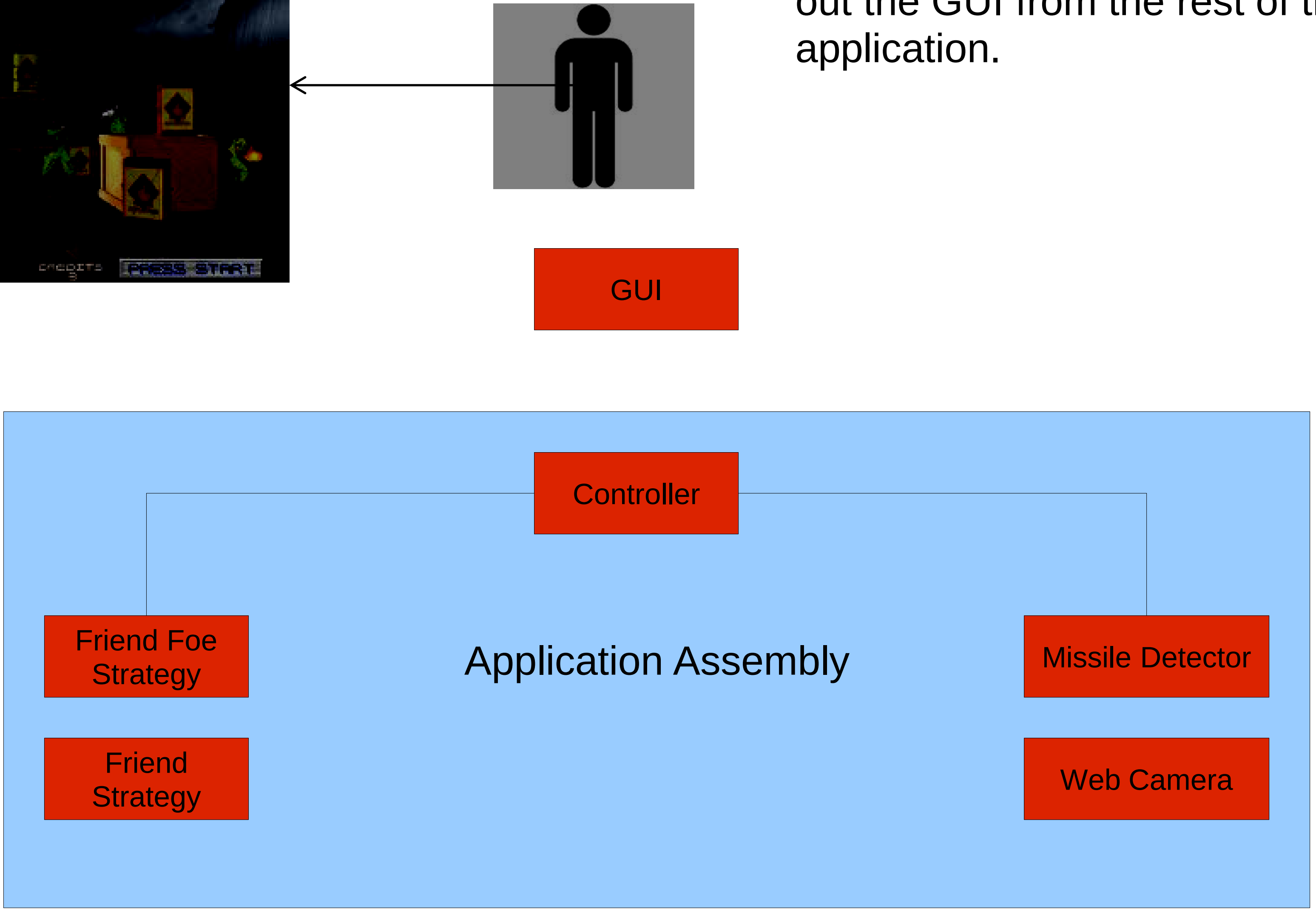
We have a general structure of the application. Everything is contained within one assembly/exe/dll



Project System

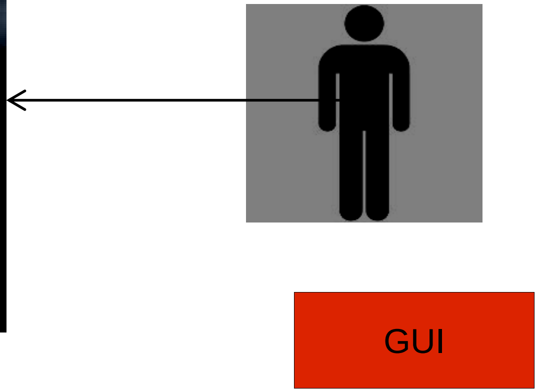
Our Target System

Let's modularize first by pulling out the GUI from the rest of the application.



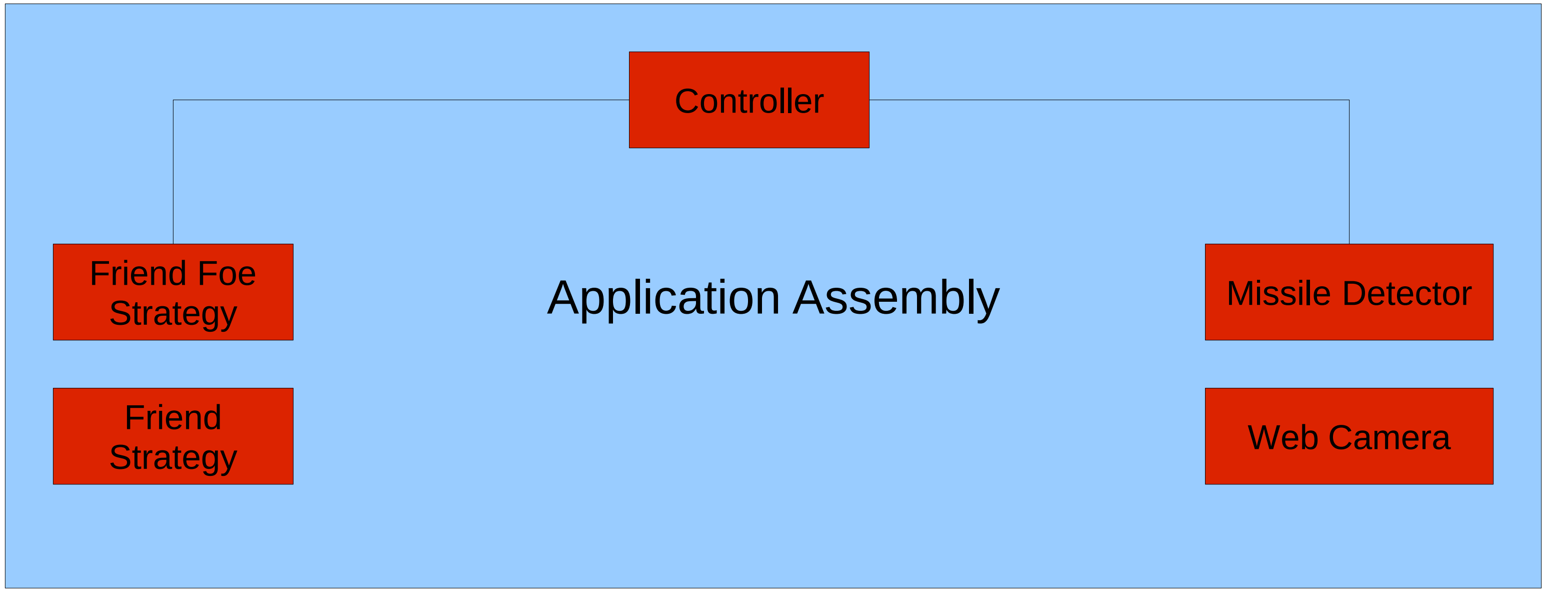
Project System

Our Target System

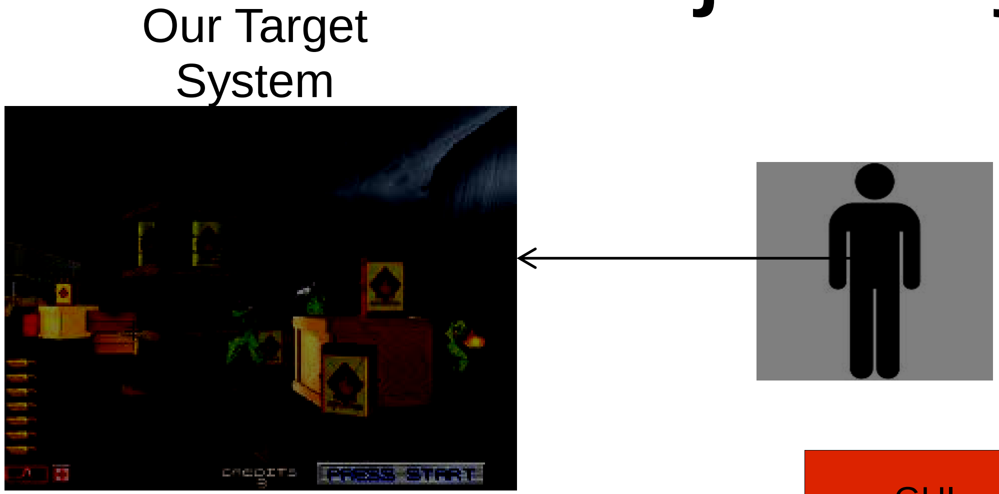


Now we're cooking with gas.

Assume that our simple model here is employing a strategy pattern already. We're using the mediator pattern to signal messages between I/O components and strategies

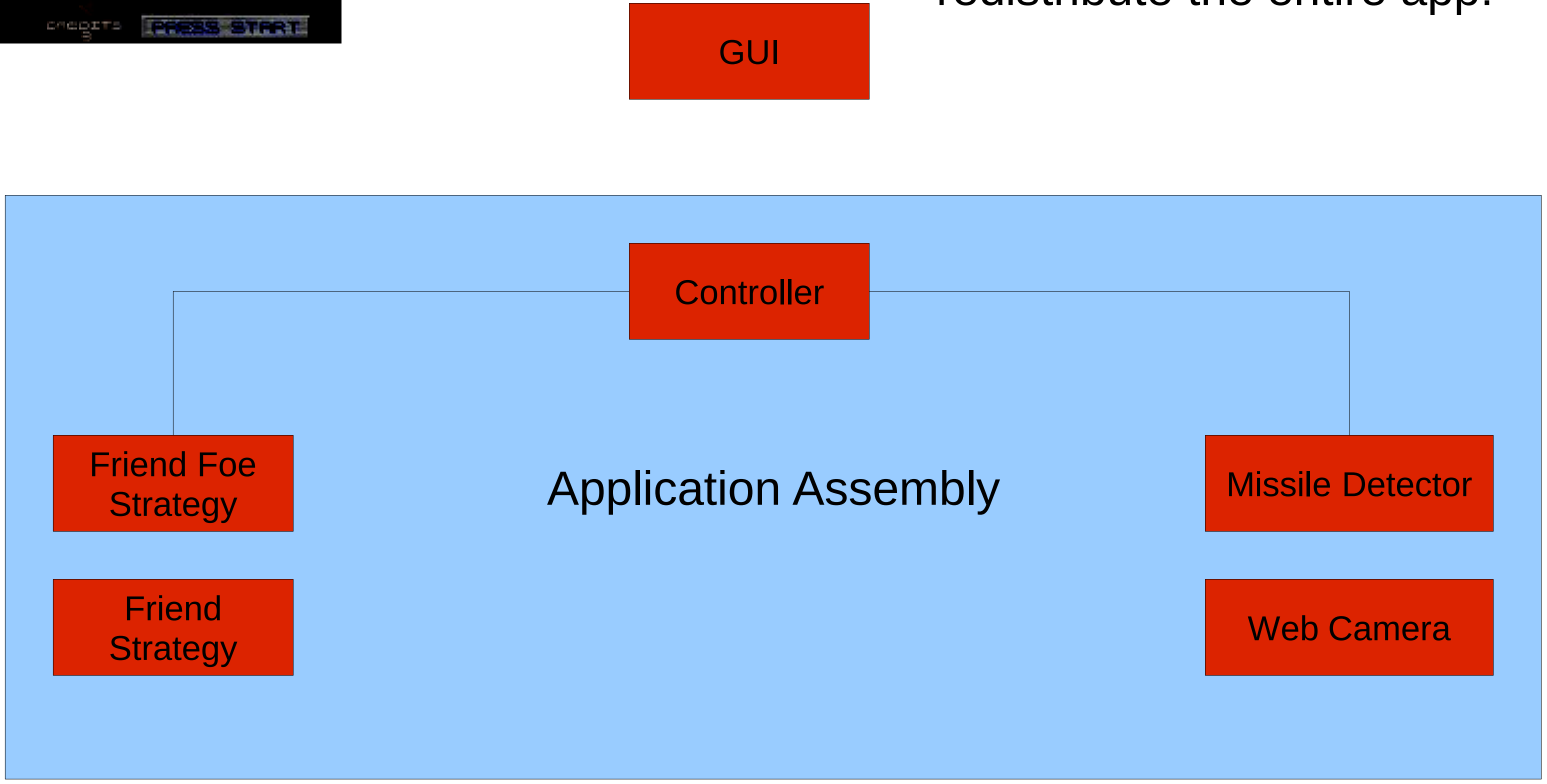


Project System



Our project manager tells us he wants to use a different camera for the image capture.

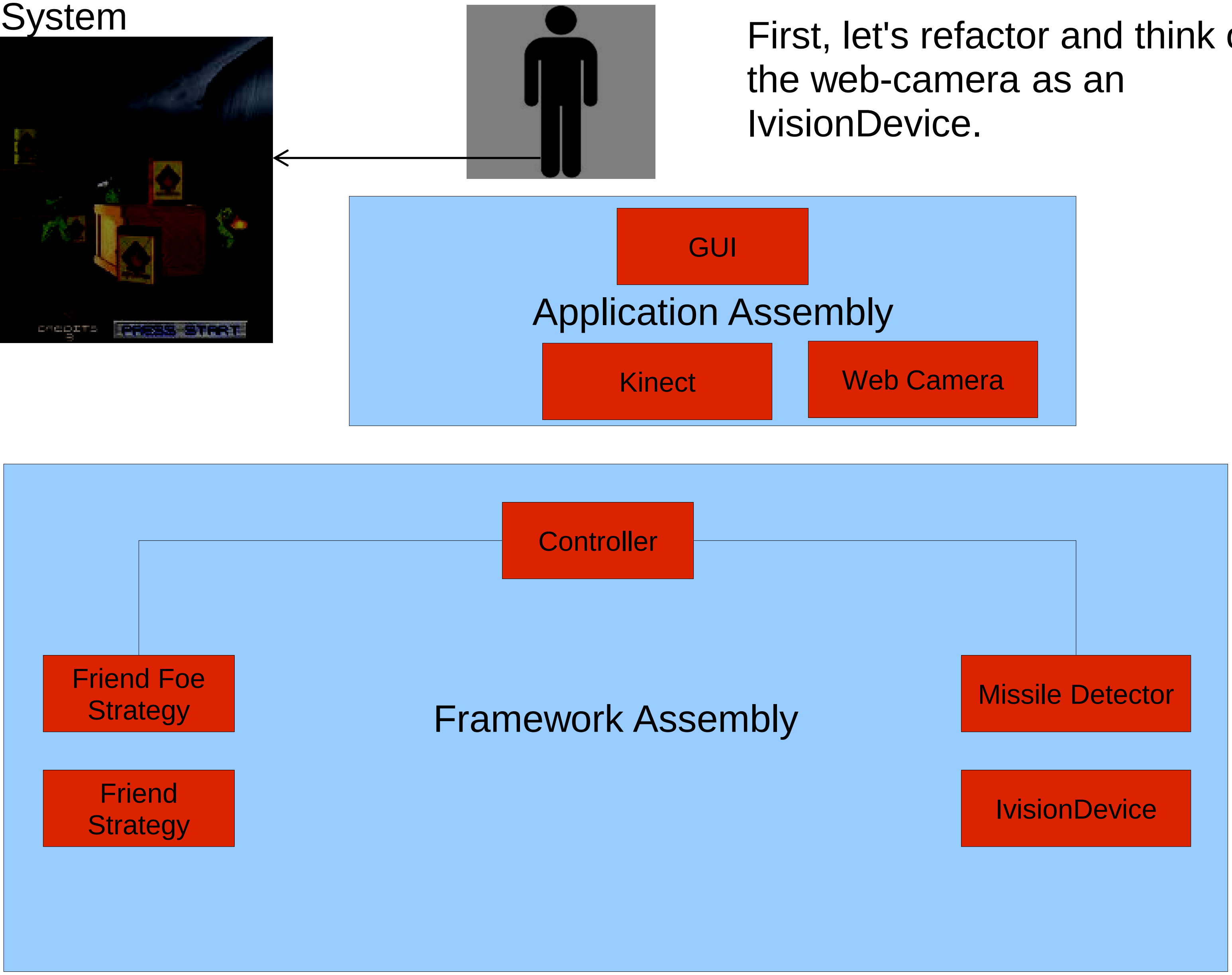
Now we have to write a module, test, recompile, and redistribute the entire app!



Project System

Our Target System

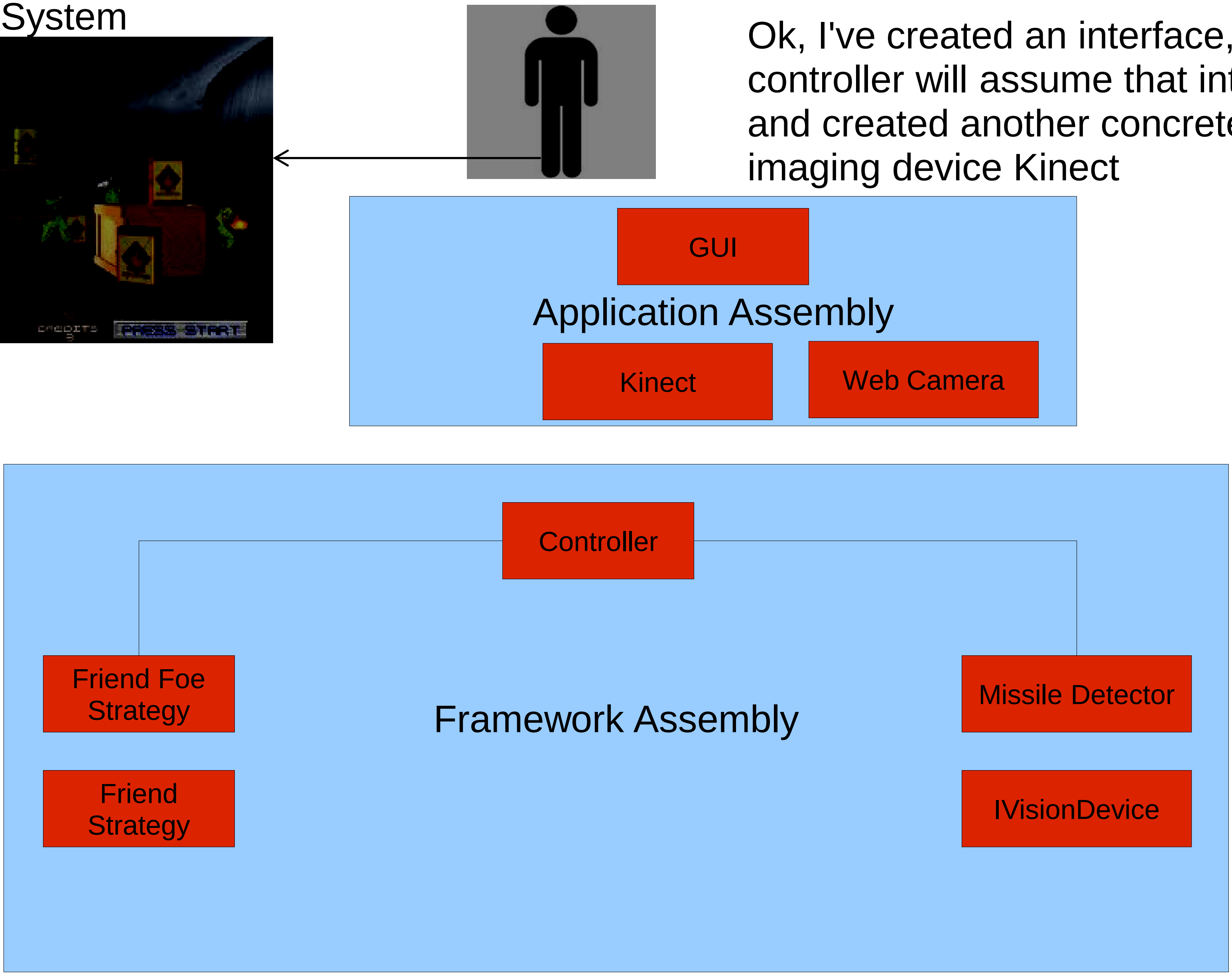
First, let's refactor and think of the web-camera as an IvisionDevice.



Project System

Our Target System

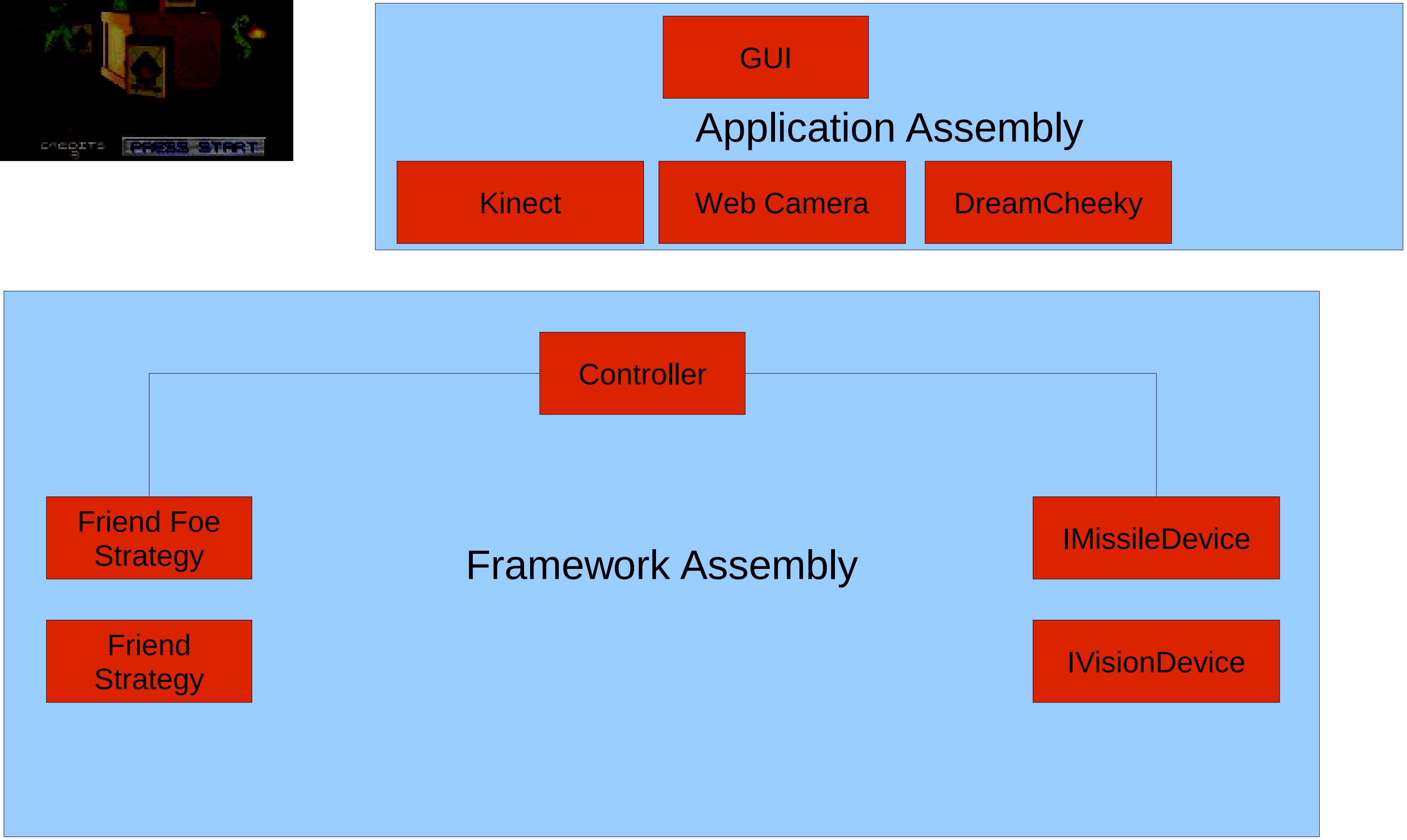
Ok, I've created an interface, the controller will assume that interface, and created another concrete imaging device Kinect



Project System

Our Target System

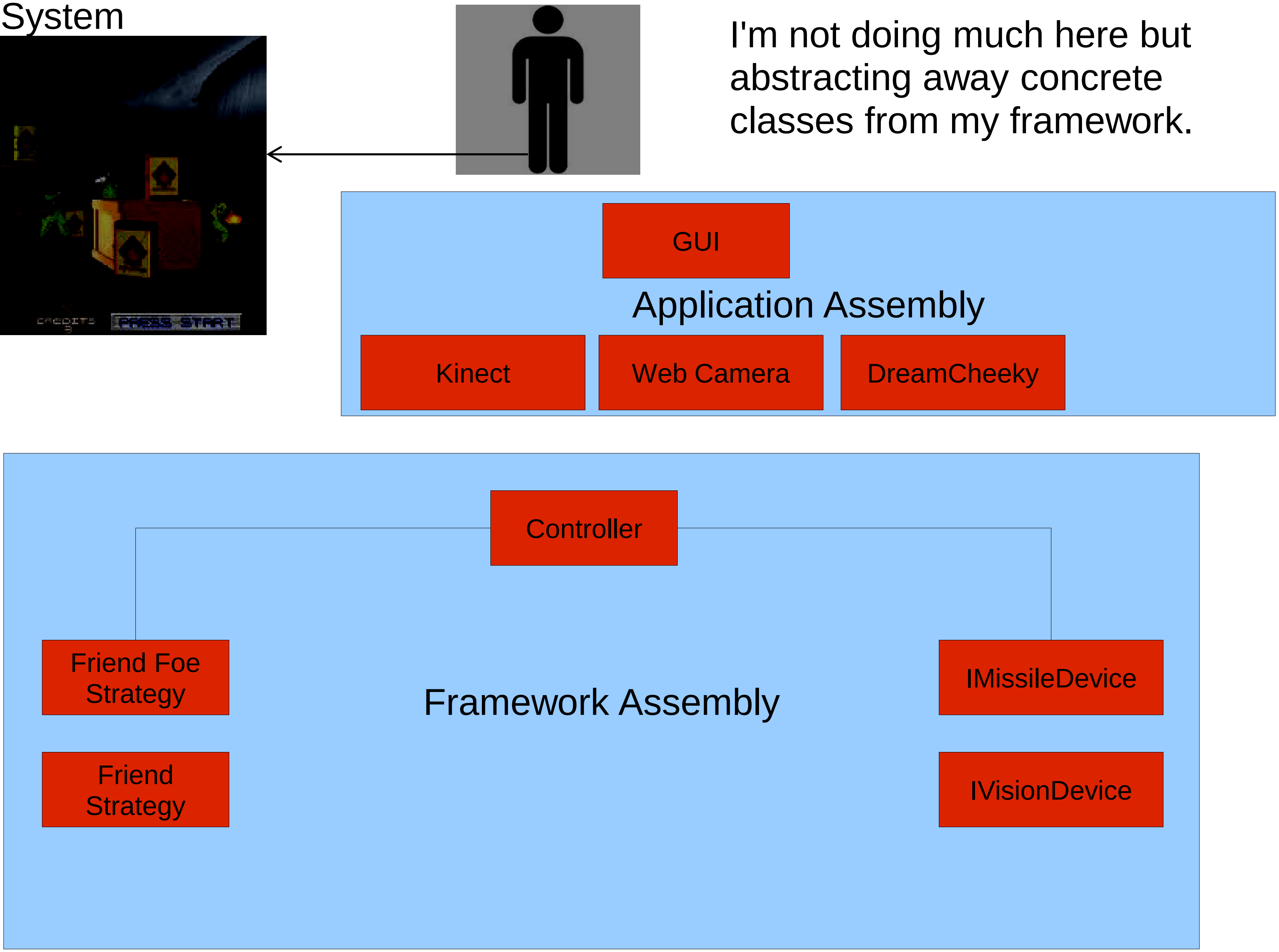
I'll do the same for the Missile Launcher.



Project System

Our Target System

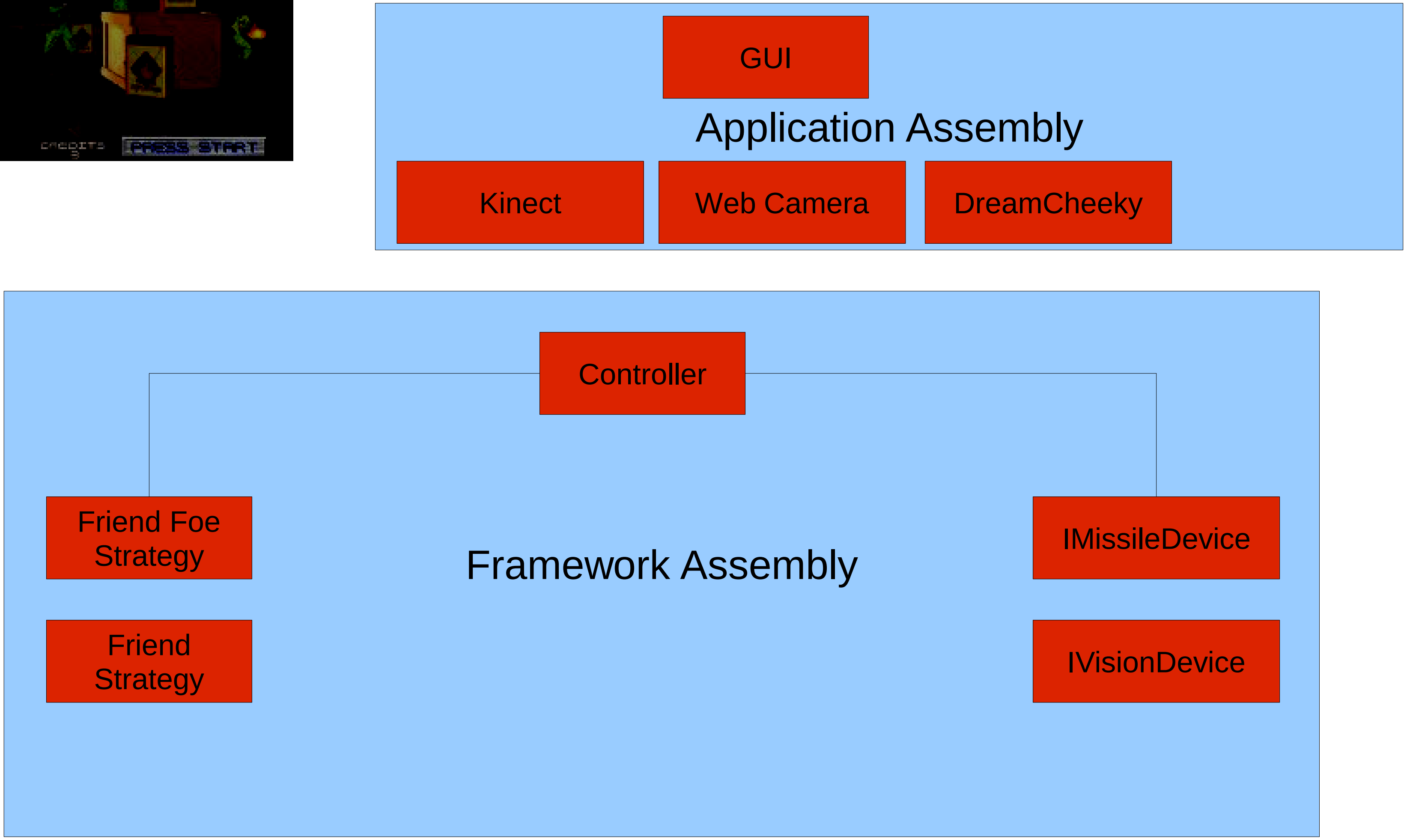
I'm not doing much here but abstracting away concrete classes from my framework.



Project System

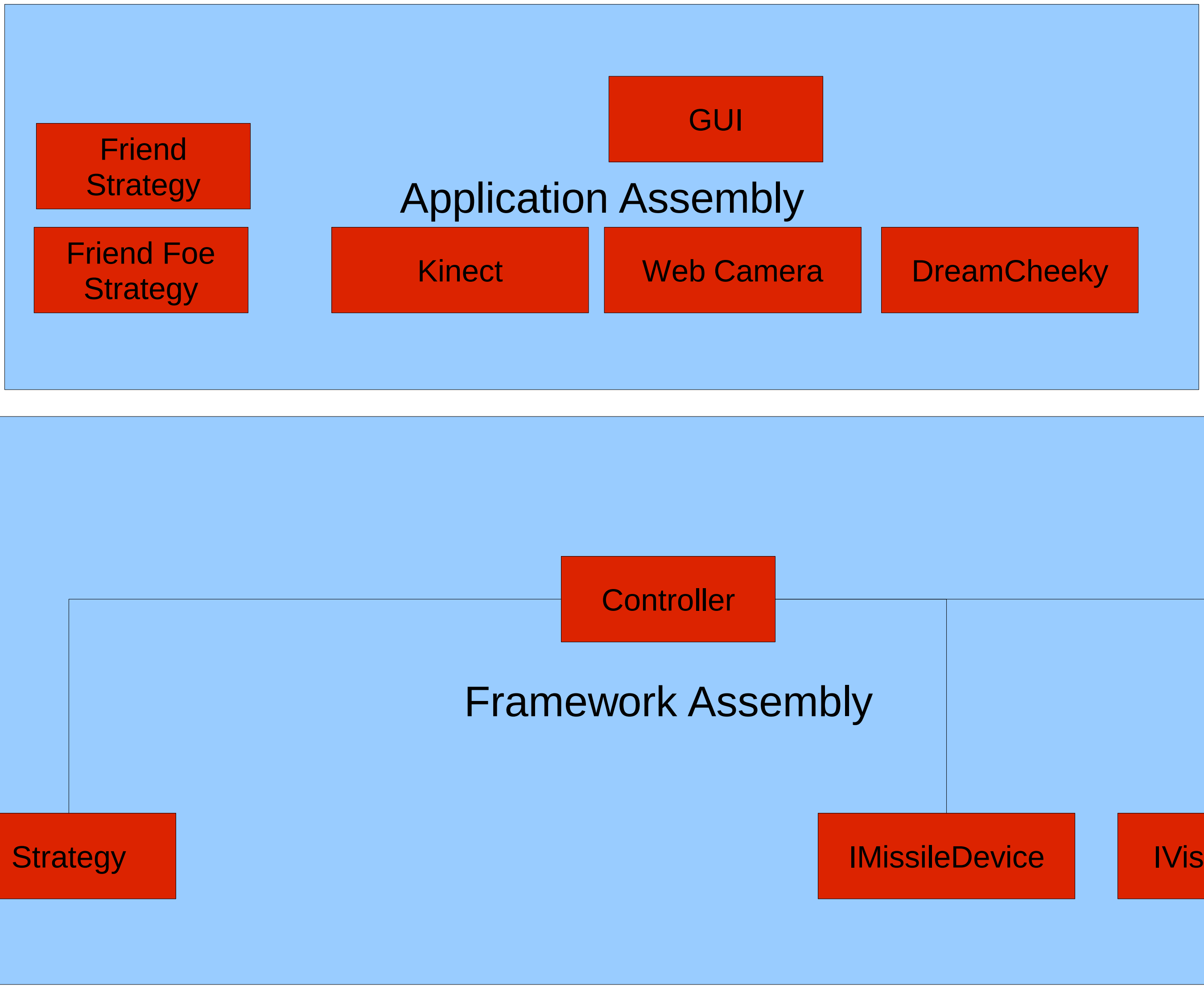
Our Target System

While we're at it, let's also abstract out the strategy pattern. I want the framework to be as generic as pragmatically possible.



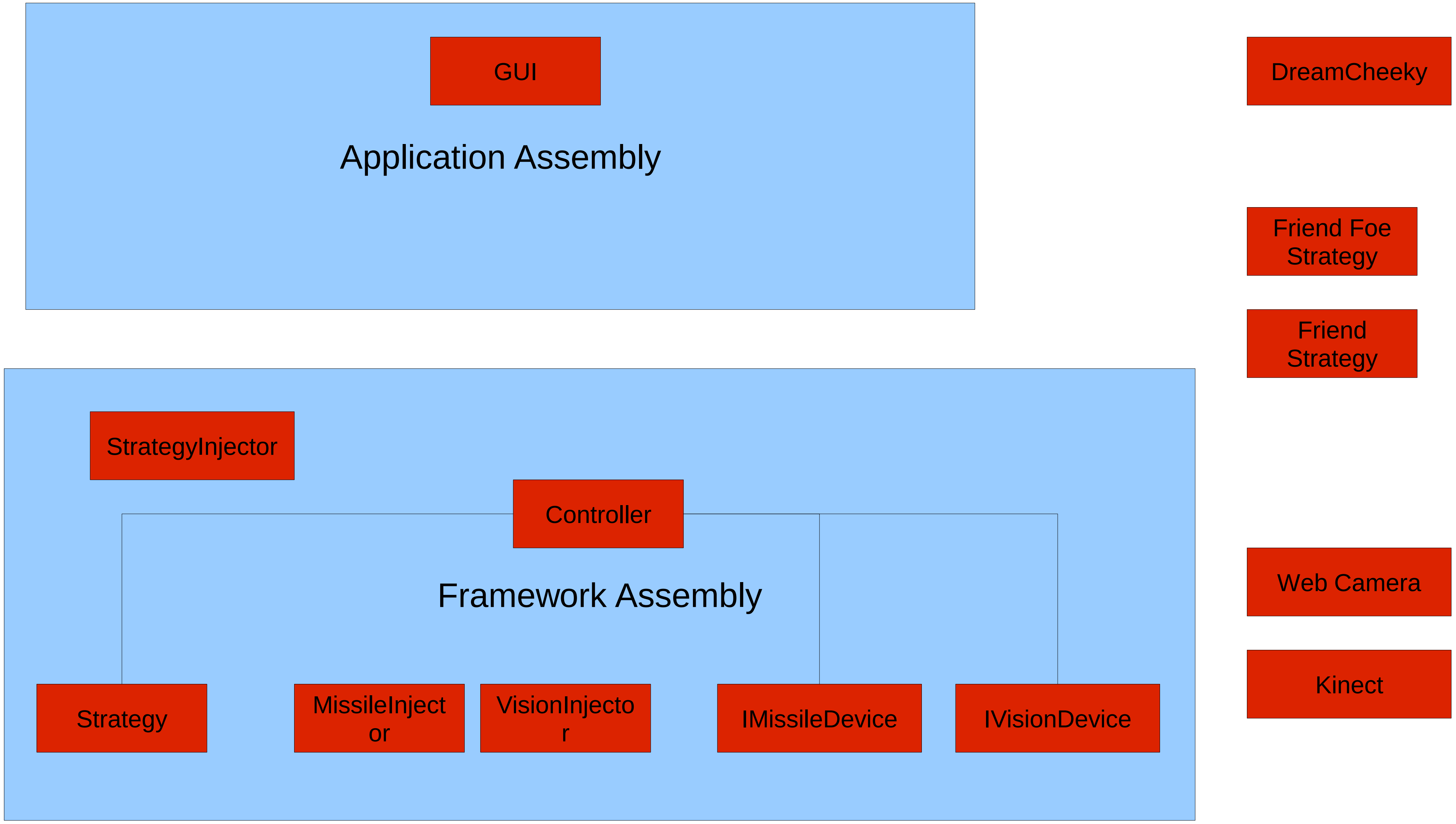
Project System

Ok, I've created basically an abstract framework that coordinates the events from my vision system, missile system, and algorithmic logic in my strategy. But at the cost of moving all of my components to the same package as my GUI



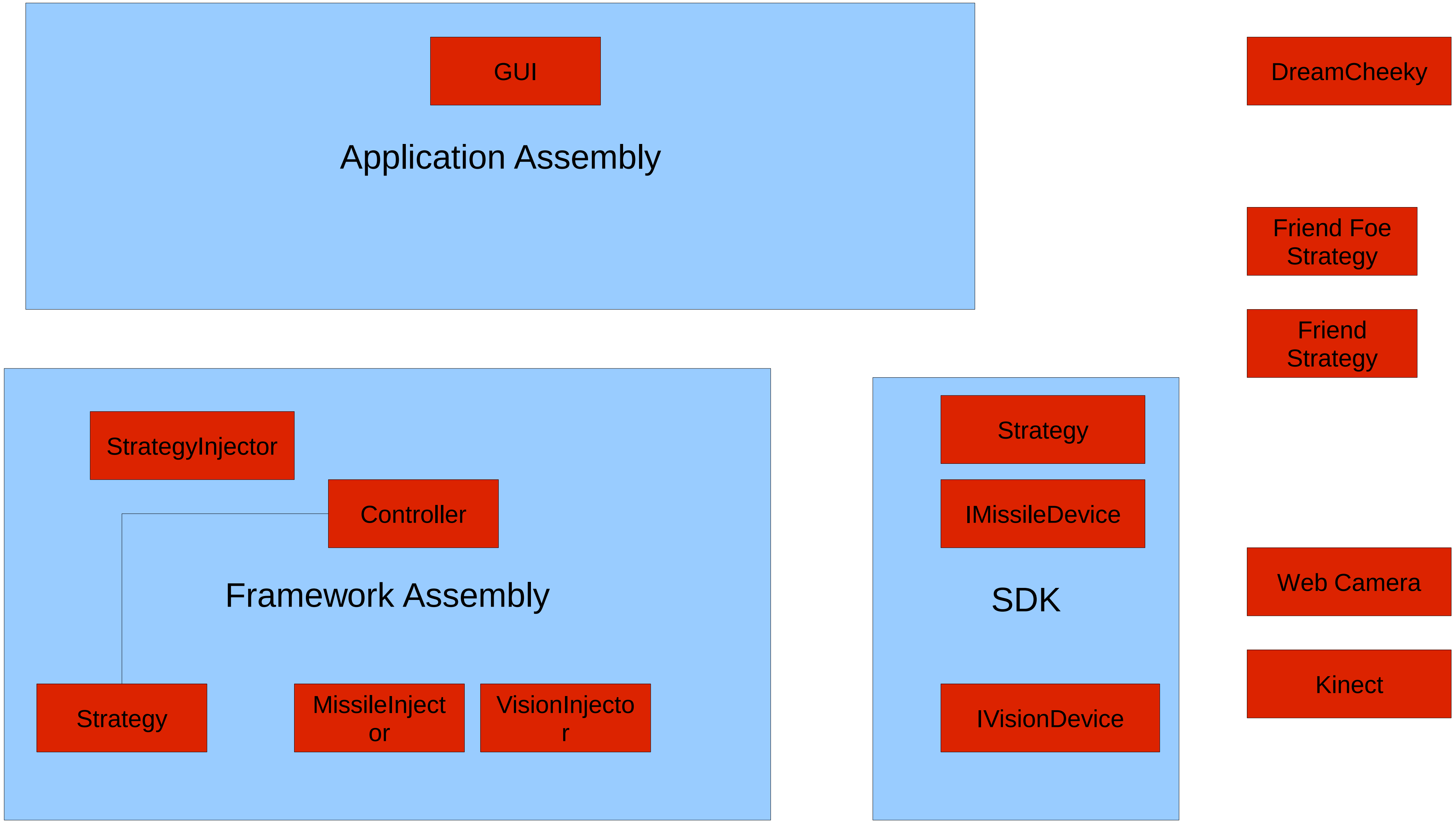
Project System

Now we want to move each concrete object into it's own dependency. Also, we'll need to make injectors (apply factory) for each interface type we want.



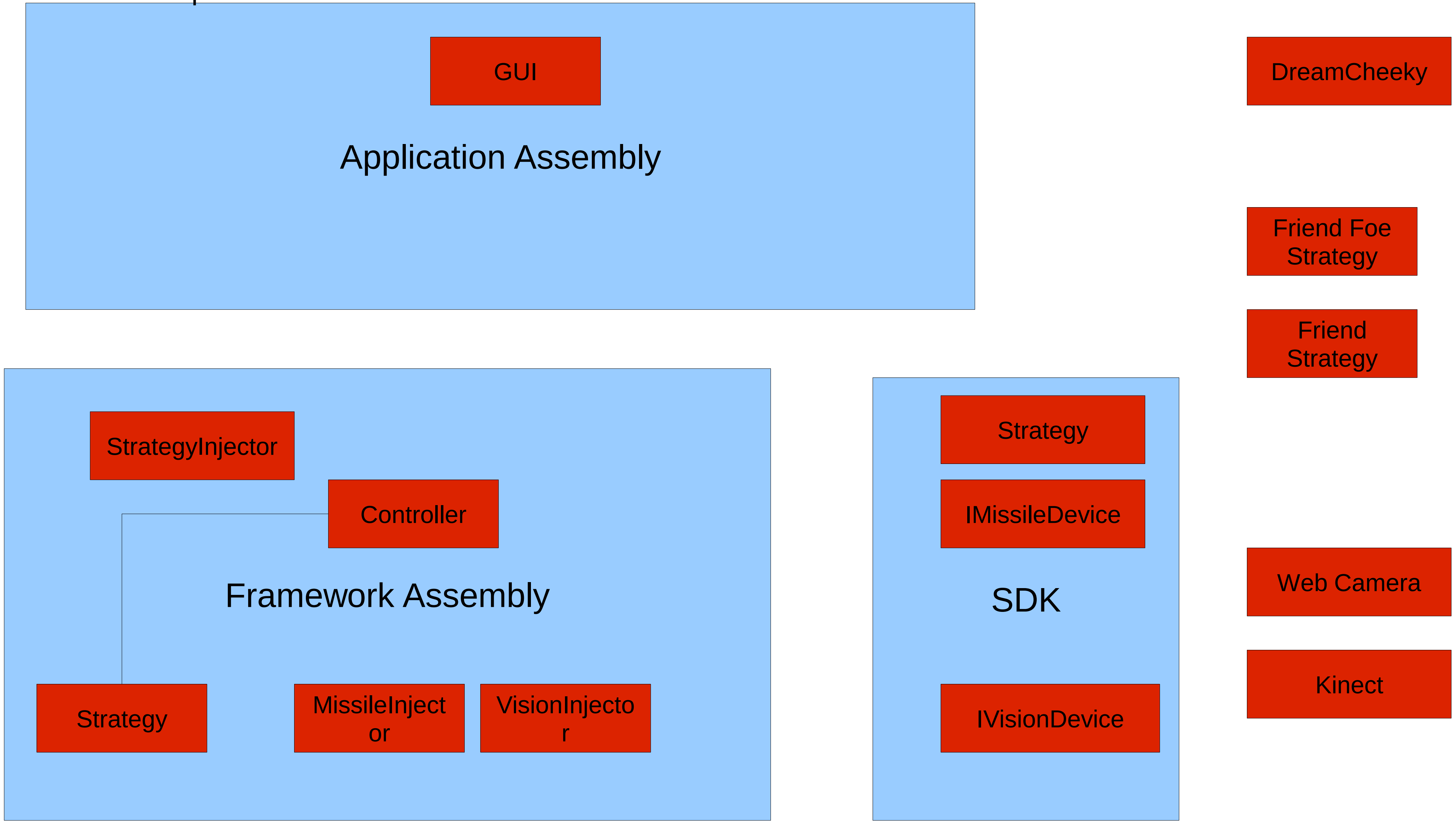
Project System

But, we should also move our interfaces into their separate assembly so that we dont have to ship our framework with stuff other dev's don't need.



Project System

So now we have separated our objects into their own assemblies. Our controller can vary independently of the GUI and concrete objects. We use the SDK as a way for us to share our interface definitions with other developers. We can extend to our hearts content.



C# and .net Plugins

Injector Code – Searching Meta-Data and Creating Plug-ins

```
/// <summary>
        /// Loads all available plugins from the file of type T.
        /// </summary>
        /// <param name="path">Path to .net assembly.</param>
        /// <returns></returns>
        public Collection<T> LoadPlugins(string path)
        {
            return LoadPlugins(Assembly.LoadFile(path));
        }
        /// <summary>
        /// Loads all available plugins from the assembly of type T.
        /// </summary>
        /// <param name="assembly"></param>
        /// <returns></returns>
        public Collection<T> LoadPlugins(Assembly assembly)
        {
                Collection<T> types = new Collection<T>();
                foreach (Type t in assembly.GetTypes())
                {
                    // Here we are checking to see if
                    if (typeof(T).IsAssignableFrom(t))
                    {
                        // You should wrap this in error handling.
                        T plugin = Activator.CreateInstance(t) as T;
                        types.Add(plugin);
                    }
                }
                return types;
        }
```

Loading plug-ins in the controller

In constructor

```
m_controller.LauncherLoaded += new
EventHandler<Plugins.PluginLoadEventArgs<GridironMaidenEngine.Launchers.ILauncher>>(m_controller_LauncherLoaded);

            m_controller.VisionDevicesLoaded += new
EventHandler<Plugins.PluginLoadEventArgs<GridironMaidenEngine.Vision.IVisionDevice>>(m_controller_VisionDevicesLoaded);


#region Plugins
         /// <summary>
         /// Loads the plugins from the path provided.
         /// </summary>
         public void LoadPlugins(string path)
         {
            GenericPluginLoader<ILauncher> launcher = new GenericPluginLoader<ILauncher>();
            Collection<ILauncher> launcherplugins = launcher.LoadPlugins(path);

            if (LauncherLoaded != null)
              {
                LauncherLoaded(this, new PluginLoadEventArgs<ILauncher>(launcherplugins));
              }

            GenericPluginLoader<IVisionDevice> vision = new GenericPluginLoader<IVisionDevice>();
            Collection<IVisionDevice> visionPlugins = vision.LoadPlugins(path);

            if (VisionDevicesLoaded != null)
              {
                VisionDevicesLoaded(this, new PluginLoadEventArgs<IVisionDevice>(visionPlugins));
              }
         }
```

In my dependent (my main form)

```
#region Plugin Event Handlers

        void m_controller_VisionDevicesLoaded(object sender, Plugins.PluginLoadEventArgs<GridironMaidenEngine.Vision.IVisionDevice> e)

        {

            m_cameras = e.Plugins;

        }

        void m_controller_LauncherLoaded(object sender, Plugins.PluginLoadEventArgs<GridironMaidenEngine.Launchers.ILauncher> e)

        {

            m_missiles = e.Plugins;

        }

        #endregion
```