Welcome

To Advance through Presentation
Use Page Up and Page Down Keys

Core Foundation Plug-ins and Bundles

Mike Ferris

Introduction

- CFPlugIn: a standard plug-in model for Carbon applications
- Mac OS X application, framework and plug-in packaging
- CFBundle: an API for dealing with application packages

CFPlugIn

- Goals and non-goals of CFPlugIn
- The CFPlugIn type
- Interfaces and Types
- Factories and Instances

CFPlugIn Goals

- Create a common model for applications that need to support plug-ins
- Easily define and expose the interfaces between applications and their plug-ins
- Provide an architecture that allows for code generation of generic plug-in code
- Allow applications and plug-ins to choose between C++ and C independently

CFPlugIn Non-Goals

- Automatically support legacy plug-in models
- Provide an “object model” for Carbon
- Replace more sophisticated object runtime based plug-in strategies such as Java beans or loadable Objective-C classes

Conversion Advantages

- Conversion of legacy plug-in models may make sense for your application
  - Plug-in developers must Carbonize anyway
  - A common plug-in model is one less thing you must teach plug-in developers
  - You might more easily leverage plug-ins written for other apps
  - Code you don’t write is code you don’t maintain

The CFPlugIn Type

- CFPlugIn represents a loadable Plug-in
- Each CFPlugIn is associated with a CFBundle
- Plug-ins are packaged with the same structure as applications and frameworks
- Once a CFPlugIn has been loaded, the Factories that it provides can be used to “instantiate” any Types that the plug-in supports

Interfaces

- Interfaces define the API between an application and its plug-ins
- Each Interface has a unique name (like “com.xyz.Imagine.ImageReaderInterface”)
- An Interface is a group of semantically related function prototypes
- In your code, an Interface is a struct containing one or more function pointers

Types

- Types are aggregations of Interfaces
- Each type has a unique name (like “com.xyz.Imagine.ImageReaderType”)
- Applications load plug-ins that implement specific Types defined by the application
- Applications instantiate Types from a plug-in
- Interfaces supported by a Type are accessed through Instances of the Type

Example—Part 1
Declaring Interfaces and Types

```
#define kImageReaderInterface
    CFSTR("com.xyz.Imagine.ImageReaderInterface");

#define kImageReaderType
    CFSTR("com.xyz.Imagine.ImageReaderType");

typedef struct _ImageReaderFtbl {
    ImageViewerImage *(*readImageWithURL)(CFPlugInInstanceRef
            instance, CFURLRef url);
} ImageReaderFtbl;
```

Example—Part 1
Declaring Interfaces and Types

```
#define kImageReaderInterface
    CFSTR("com.xyz.Imagine.ImageReaderInterface");

#define kImageReaderType
    CFSTR("com.xyz.Imagine.ImageReaderType");

typedef struct _ImageReaderFtbl {
    ImageViewerImage *(*readImageWithURL)(CFPlugInInstanceRef
            instance, CFURLRef url);
} ImageReaderFtbl;
```

Example—Part 1
Declaring Interfaces and Types

```
#define kImageReaderInterface
    CFSTR("com.xyz.Imagine.ImageReaderInterface");

#define kImageReaderType
    CFSTR("com.xyz.Imagine.ImageReaderType");

typedef struct _ImageReaderFtbl {
    ImageViewerImage *(*readImageWithURL)(CFPlugInInstanceRef
            instance, CFURLRef url);
} ImageReaderFtbl;
```

Factories

- Factories are functions provided by a plug-in that can instantiate one or more Types
- Each Factory has a unique name (like “com.acme.Imagine.GIFReaderFactory”)
- Each plug-in must provide a Factory for each Type that it supports
- An application can find all the Factories that can instantiate a given Type

Factory Registration

- Factories are registered either statically or dynamically
- Most plug-ins use static registration
- Meta-data packaged with the plug-in is used for static registration
- A registration function within the plug-in is used for dynamic registration

Example—Part 2
Registering a Factory

```
{
    CFPlugInDynamicRegistration = NO;
    CFPlugInFactories = {
        "com.acme.Imagine.GIFReaderFactory" = MyFactory;
    };
    CFPlugInTypes = {
        "com.xyz.Imagine.ImageReaderType" =
                ("com.acme.Imagine.GIFReaderFactory");
    };
}
```

Example—Part 2
Registering a Factory

```
{
    CFPlugInDynamicRegistration = NO;
    CFPlugInFactories = {
        "com.acme.Imagine.GIFReaderFactory" = MyFactory;
    };
    CFPlugInTypes = {
        "com.xyz.Imagine.ImageReaderType" =
                ("com.acme.Imagine.GIFReaderFactory");
    };
}
```

Example—Part 2
Registering a Factory

```
{
    CFPlugInDynamicRegistration = NO;
    CFPlugInFactories = {
        "com.acme.Imagine.GIFReaderFactory" = MyFactory;
    };
    CFPlugInTypes = {
        "com.xyz.Imagine.ImageReaderType" =
                ("com.acme.Imagine.GIFReaderFactory");
    };
}
```

Instances

- CFPlugInInstance represents a single Instance of a Type
- CFPlugInInstances are created by Factories
- They provide access to the Interfaces supported by the Type
- An application can create many Instances of a single Type
- CFPlugInInstances hold any per-instance data that the plug-in needs

Example—Part 3
Implementing an Interface

```
static ImageViewerImage
    *myReadImageWithURL(CFPlugInInstanceRef
            instance, CFURLRef url) {
    /* Code to read in the image from the given URL */
}

static ImageReaderFtbl imageReaderFtbl { myReadImageWithURL };

static Boolean myGetInterface(CFPlugInInstanceRef instance,
            CFStringRef interfaceName, void **ftbl) {
    *ftbl = NULL;
    if (CFEqual(interfaceName, kImageReaderInterface)) {
        *ftbl = &imageReaderFtbl;
    }
    return (*ftbl ? TRUE : FALSE);
}
```

Example—Part 3
Implementing an Interface

```
static ImageViewerImage
    *myReadImageWithURL(CFPlugInInstanceRef
            instance, CFURLRef url) {
    /* Code to read in the image from the given URL */
}

static ImageReaderFtbl imageReaderFtbl { myReadImageWithURL };

static Boolean myGetInterface(CFPlugInInstanceRef instance,
            CFStringRef interfaceName, void **ftbl) {
    *ftbl = NULL;
    if (CFEqual(interfaceName, kImageReaderInterface)) {
        *ftbl = &imageReaderFtbl;
    }
    return (*ftbl ? TRUE : FALSE);
}
```

Example—Part 3
Implementing an Interface

```
static ImageViewerImage
    *myReadImageWithURL(CFPlugInInstanceRef
            instance, CFURLRef url) {
    /* Code to read in the image from the given URL */
}

static ImageReaderFtbl imageReaderFtbl { myReadImageWithURL };

static Boolean myGetInterface(CFPlugInInstanceRef instance,
            CFStringRef interfaceName, void **ftbl) {
    *ftbl = NULL;
    if (CFEqual(interfaceName, kImageReaderInterface)) {
        *ftbl = &imageReaderFtbl;
    }
    return (*ftbl ? TRUE : FALSE);
}
```

Example—Part 4
Implementing a Factory Function

```
#define kGIFImageReaderFactory
        CFSTR("com.acme.Imagine.GIFReaderFactory")

CFPlugInInstanceRef MyFactory(CFAllocatorRef allocator,
                CFStringRef typeName) {
    if (CFEqual(typeName, kImageReaderType)) {
        return
    CFPlugInInstanceCreateWithInstanceDataSize(allocator,
                    0, NULL, kGIFImageReaderFactory , myGetInterface);
    } else {
        return NULL;
    }
}
```

Example—Part 4
Implementing a Factory Function

```
#define kGIFImageReaderFactory
        CFSTR("com.acme.Imagine.GIFReaderFactory")

CFPlugInInstanceRef MyFactory(CFAllocatorRef allocator,
                CFStringRef typeName) {
    if (CFEqual(typeName, kImageReaderType)) {
        return
    CFPlugInInstanceCreateWithInstanceDataSize(allocator,
                    0, NULL, kGIFImageReaderFactory , myGetInterface);
    } else {
        return NULL;
    }
}
```

Lifetime Issues

- CFPlugIn and CFPlugInInstance are reference counted like all CF types
- When a CFPlugIn's reference count drops to zero, its code is unloaded and the CFPlugIn is freed
- Each CFPlugInInstance implicitly retains the CFPlugIn it was instantiated from, so a CFPlugIn will not unload as long as any instances that it created still exist

Example—Part 5
Loading and Using a Plug-in

```
CFPlugInRef plugin = CFPlugInCreate(NULL, someURL);
CFArrayRef factories =
        CFPlugInFindFactoriesForPlugInType(kImageReaderType);
if (factories && (CFArrayGetCount(factories) > 0)) {
    CFStringRef factoryName = CFArrayGetValueAtIndex(factories, 0);
    CFPlugInInstanceRef instance = CFPlugInInstanceCreate(NULL,
            factoryName, kImageReaderType);
    ImageReaderFtbl *ftbl;
    CFPlugInInstanceGetInterfaceFunctionTable(instance,
            kImageReaderInterface, (void **)&ftbl);
    theImage = ftbl->readImageWithURL(instance, someImageURL);
    CFRelease(instance);
}
CFRelease(plugin);
```

Example—Part 5
Loading and Using a Plug-in

```
CFPlugInRef plugin = CFPlugInCreate(NULL, someURL);
CFArrayRef factories =
        CFPlugInFindFactoriesForPlugInType(kImageReaderType);
if (factories && (CFArrayGetCount(factories) > 0)) {
    CFStringRef factoryName = CFArrayGetValueAtIndex(factories, 0);
    CFPlugInInstanceRef instance = CFPlugInInstanceCreate(NULL,
            factoryName, kImageReaderType);
    ImageReaderFtbl *ftbl;
    CFPlugInInstanceGetInterfaceFunctionTable(instance,
            kImageReaderInterface, (void **)&ftbl);
    theImage = ftbl->readImageWithURL(instance, someImageURL);
    CFRelease(instance);
}
CFRelease(plugin);
```

Example—Part 5
Loading and Using a Plug-in

```
CFPlugInRef plugin = CFPlugInCreate(NULL, someURL);
CFArrayRef factories =
        CFPlugInFindFactoriesForPlugInType(kImageReaderType);
if (factories && (CFArrayGetCount(factories) > 0)) {
    CFStringRef factoryName = CFArrayGetValueAtIndex(factories, 0);
    CFPlugInInstanceRef instance = CFPlugInInstanceCreate(NULL,
            factoryName, kImageReaderType);
    ImageReaderFtbl *ftbl;
    CFPlugInInstanceGetInterfaceFunctionTable(instance,
            kImageReaderInterface, (void **)&ftbl);
    theImage = ftbl->readImageWithURL(instance, someImageURL);
    CFRelease(instance);
}
CFRelease(plugin);
```

Example—Part 5
Loading and Using a Plug-in

```
CFPlugInRef plugin = CFPlugInCreate(NULL, someURL);
CFArrayRef factories =
        CFPlugInFindFactoriesForPlugInType(kImageReaderType);
if (factories && (CFArrayGetCount(factories) > 0)) {
    CFStringRef factoryName = CFArrayGetValueAtIndex(factories, 0);
    CFPlugInInstanceRef instance = CFPlugInInstanceCreate(NULL,
            factoryName, kImageReaderType);
    ImageReaderFtbl *ftbl;
    CFPlugInInstanceGetInterfaceFunctionTable(instance,
            kImageReaderInterface, (void **)&ftbl);
    theImage = ftbl->readImageWithURL(instance, someImageURL);
    CFRelease(instance);
}
CFRelease(plugin);
```

Example—Part 5
Loading and Using a Plug-in

```
CFPlugInRef plugin = CFPlugInCreate(NULL, someURL);
CFArrayRef factories =
        CFPlugInFindFactoriesForPlugInType(kImageReaderType);
if (factories && (CFArrayGetCount(factories) > 0)) {
    CFStringRef factoryName = CFArrayGetValueAtIndex(factories, 0);
    CFPlugInInstanceRef instance = CFPlugInInstanceCreate(NULL,
            factoryName, kImageReaderType);
    ImageReaderFtbl *ftbl;
    CFPlugInInstanceGetInterfaceFunctionTable(instance,
            kImageReaderInterface, (void **)&ftbl);
    theImage = ftbl->readImageWithURL(instance, someImageURL);
    CFRelease(instance);
}
CFRelease(plugin);
```

Example—Part 5
Loading and Using a Plug-in

```
CFPlugInRef plugin = CFPlugInCreate(NULL, someURL);
CFArrayRef factories =
        CFPlugInFindFactoriesForPlugInType(kImageReaderType);
if (factories && (CFArrayGetCount(factories) > 0)) {
    CFStringRef factoryName = CFArrayGetValueAtIndex(factories, 0);
    CFPlugInInstanceRef instance = CFPlugInInstanceCreate(NULL,
            factoryName, kImageReaderType);
    ImageReaderFtbl *ftbl;
    CFPlugInInstanceGetInterfaceFunctionTable(instance,
            kImageReaderInterface, (void **)&ftbl);
    theImage = ftbl->readImageWithURL(instance, someImageURL);
    CFRelease(instance);
}
CFRelease(plugin);
```

Example—Part 5
Loading and Using a Plug-in

```
CFPlugInRef plugin = CFPlugInCreate(NULL, someURL);
CFArrayRef factories =
        CFPlugInFindFactoriesForPlugInType(kImageReaderType);
if (factories && (CFArrayGetCount(factories) > 0)) {
    CFStringRef factoryName = CFArrayGetValueAtIndex(factories, 0);
    CFPlugInInstanceRef instance = CFPlugInInstanceCreate(NULL,
            factoryName, kImageReaderType);
    ImageReaderFtbl *ftbl;
    CFPlugInInstanceGetInterfaceFunctionTable(instance,
            kImageReaderInterface, (void **)&ftbl);
    theImage = ftbl->readImageWithURL(instance, someImageURL);
    CFRelease(instance);
}
CFRelease(plugin);
```

Demo

CFPlugIn

Code Generation

Application Packaging

- Motivation and goals
- What it is used for
- Layout of a package

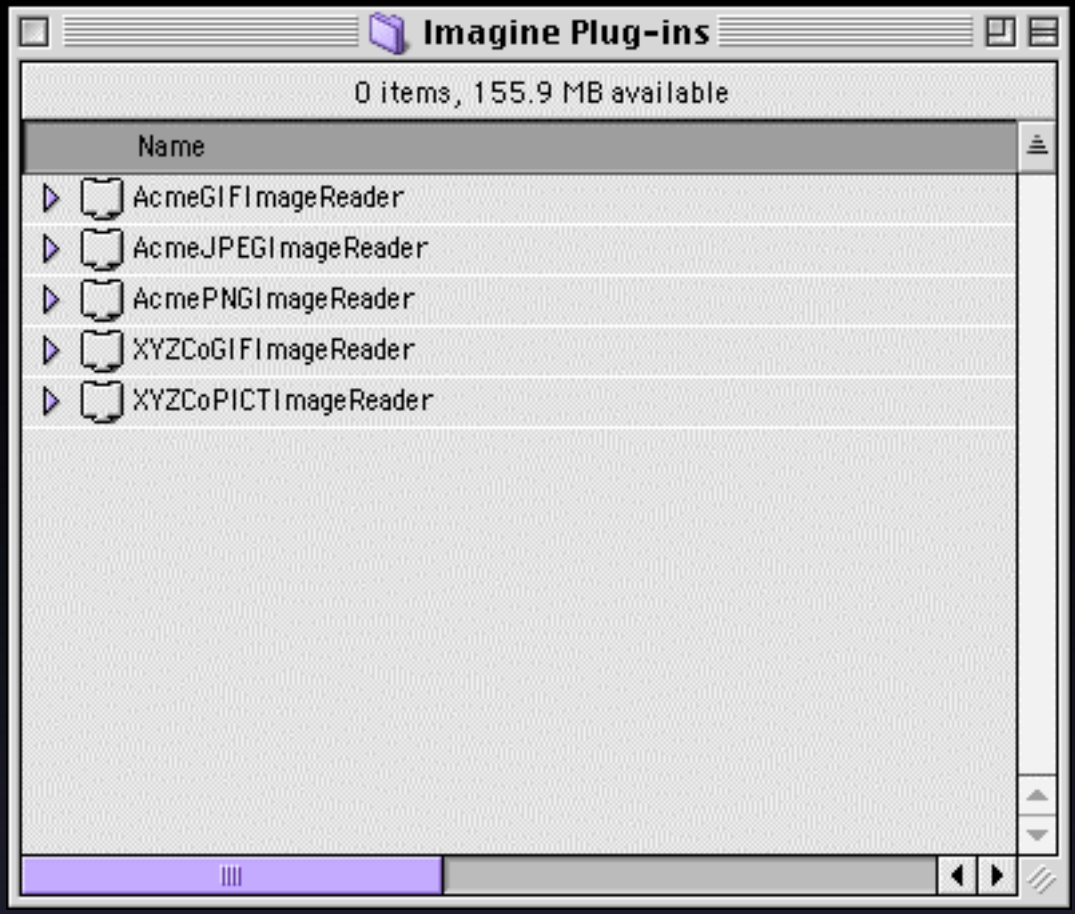
App Packaging Goals

- A single Finder item represents an application or shared library or plug-in
- Support for multiple-localization
- Provide a place to store supporting data
    - Resources
    - Files (Help, Images, Headers, etc…)
    - Executables for different platforms
    - Required libraries or loadable code

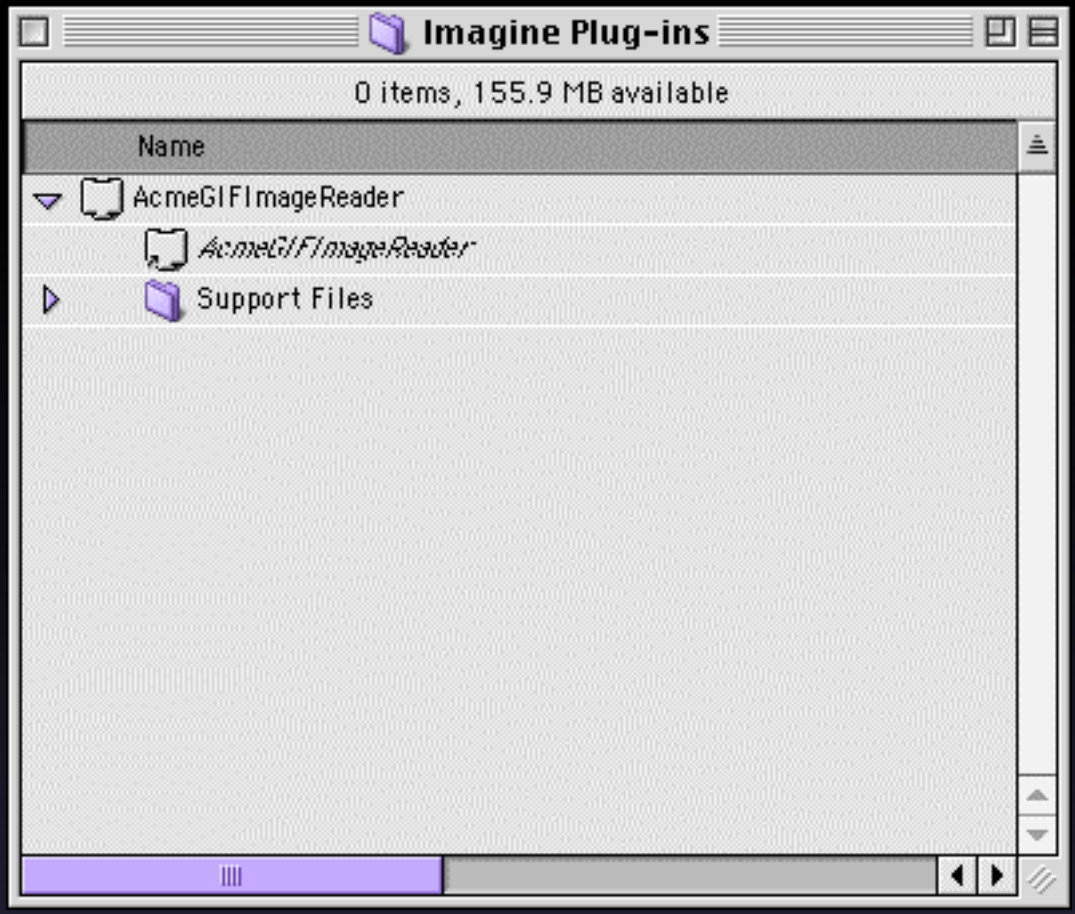
App Packaging Uses

- The same packaging structure is used for all kinds of code packages
  - Applications
  - Frameworks (shared libraries)
  - Plug-ins
  - Custom uses

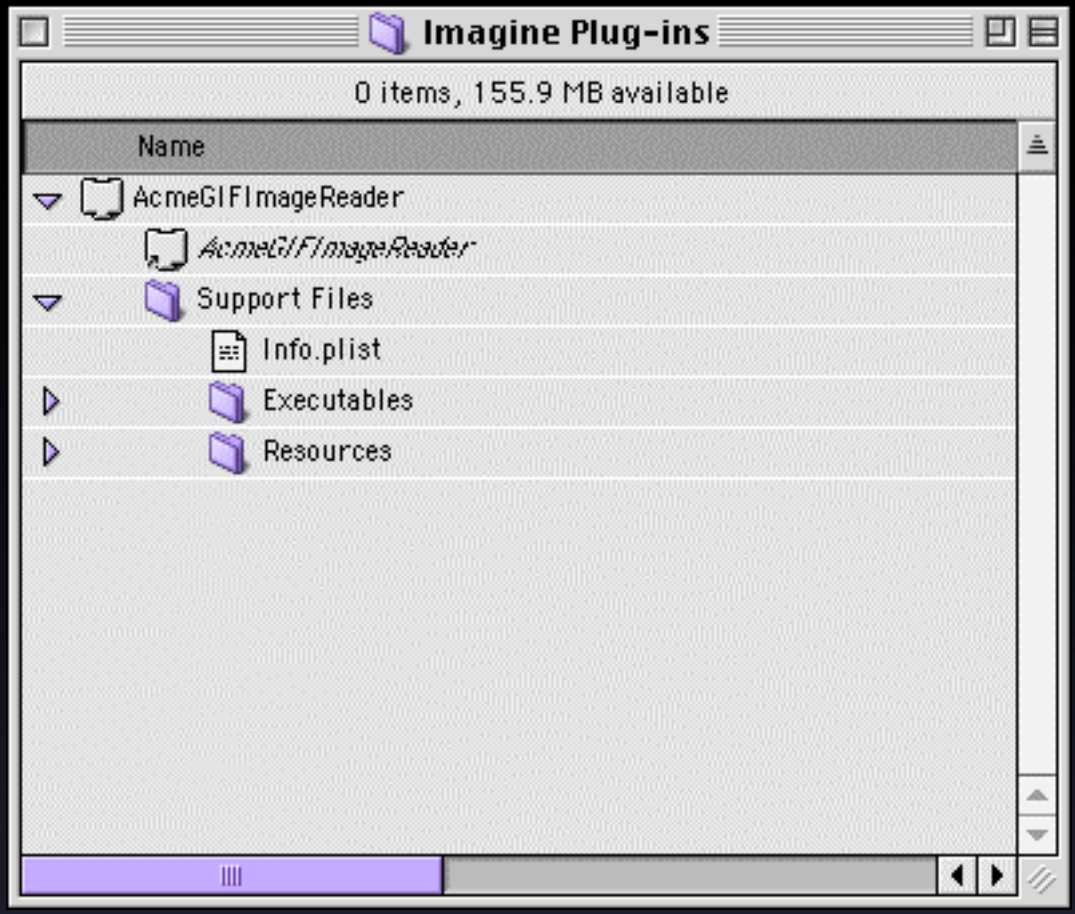
App Package Layout



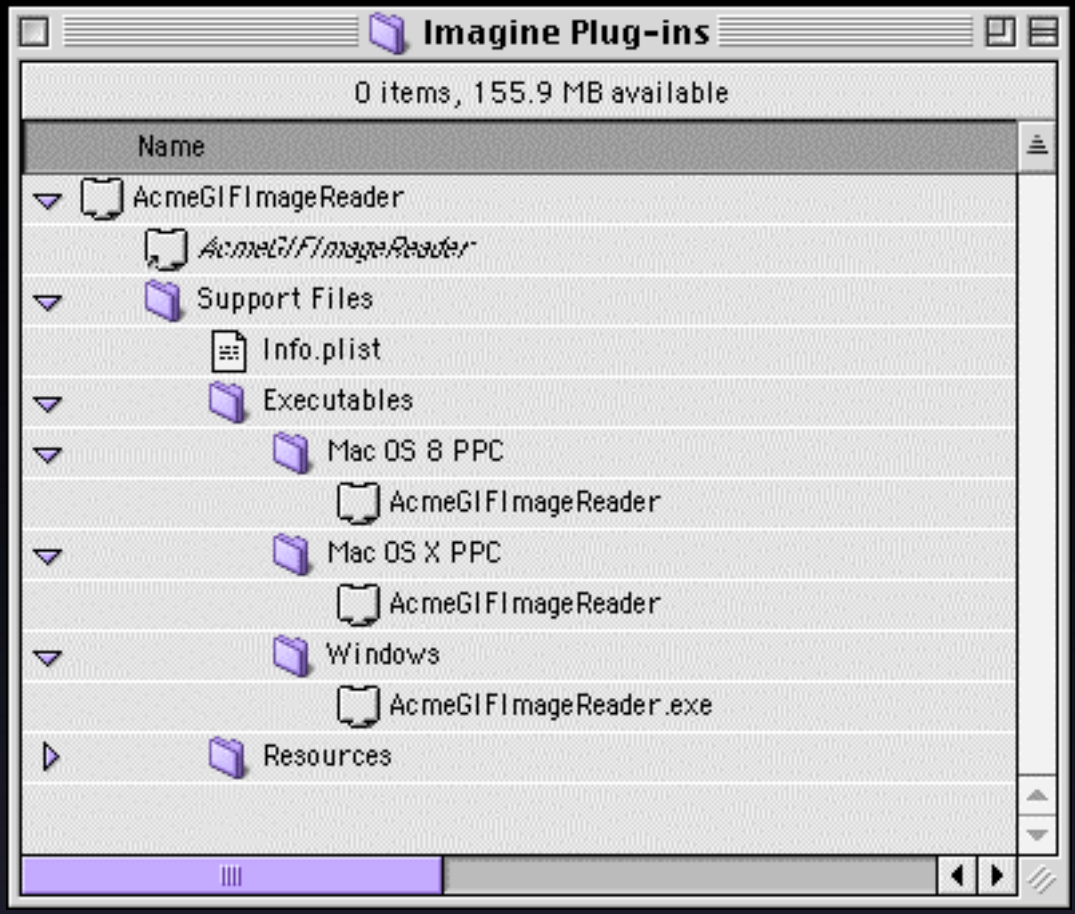
App Package Layout



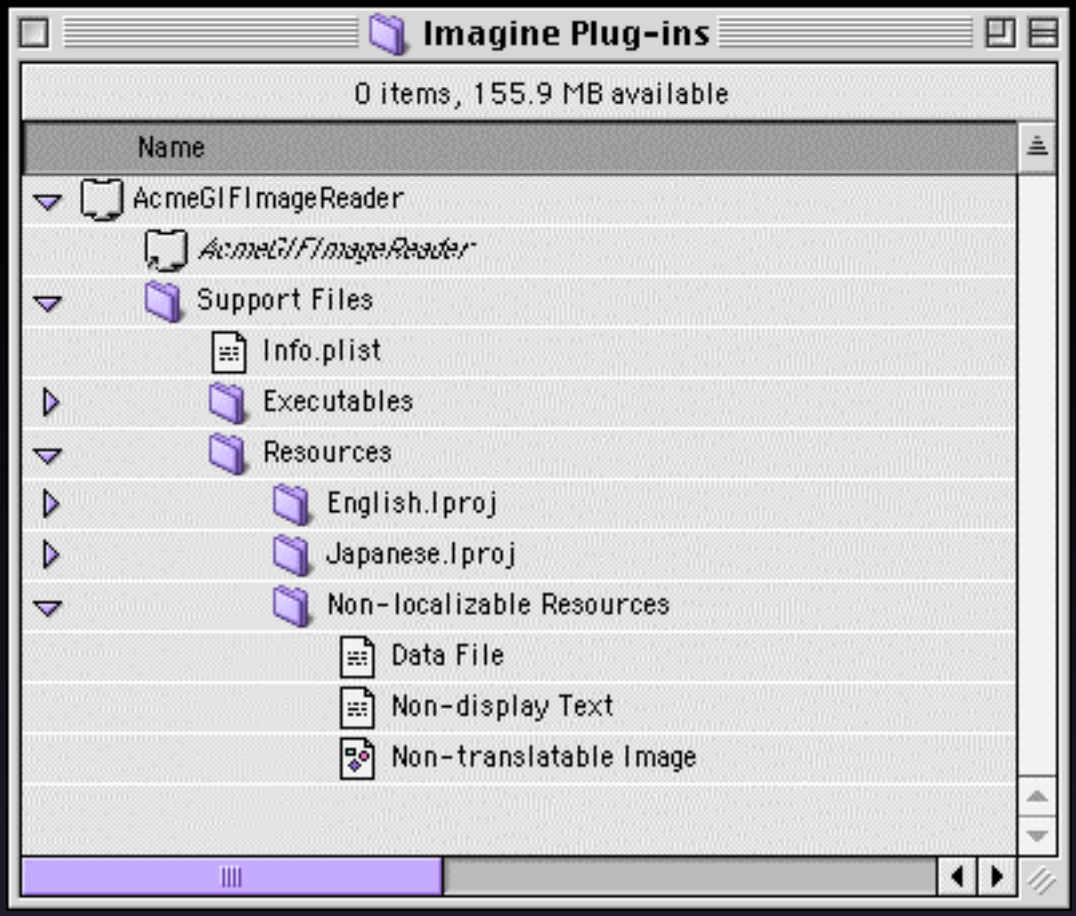
App Package Layout



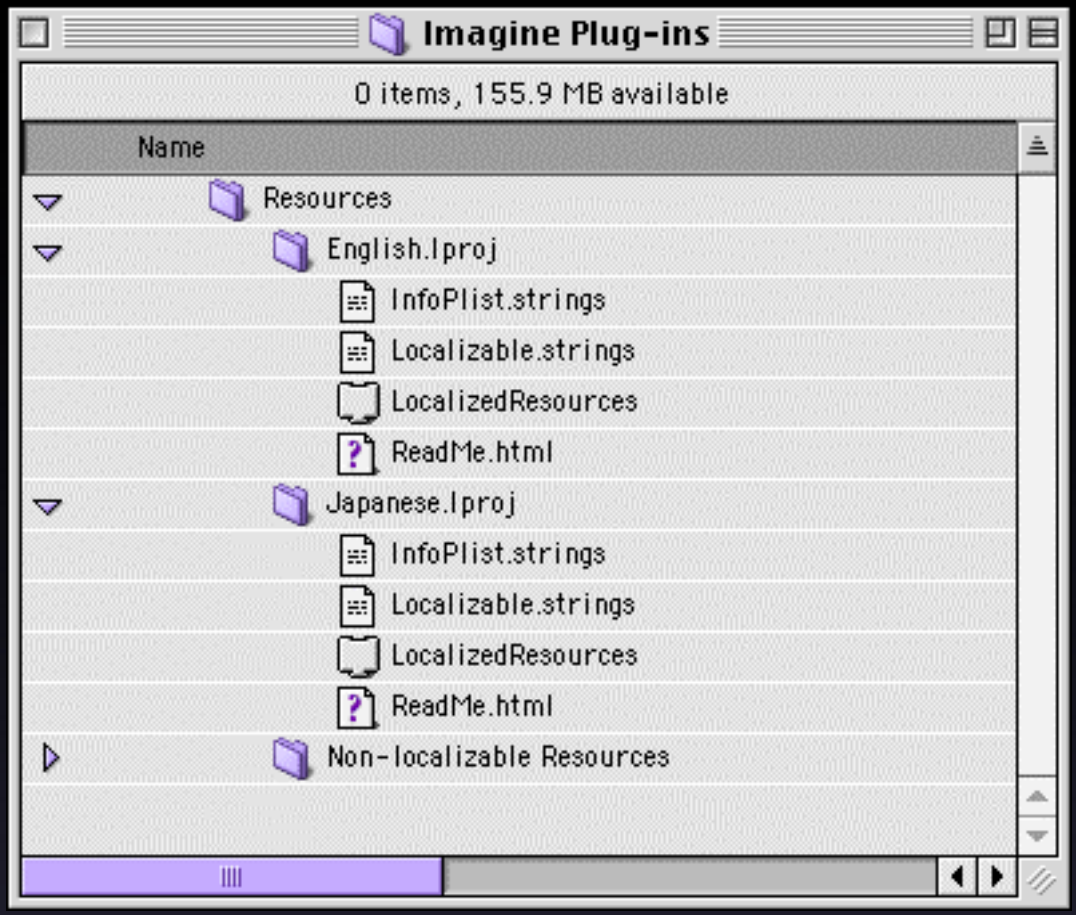
App Package Layout



App Package Layout



App Package Layout



CFBundle

- Goals
- Info.plist
- Resources—files
- Resources—forks
- Dynamic code loading

CFBundle Goals

- Provide an API for dealing with the new packaging structure
- Automatically support using the correct localization for multiply-localized bundles
- Provide an API which insulates developers from the specifics of code loading and symbol lookup on various platforms

Info.plist

- Each bundle contains an Info.plist file which contains meta-data about the bundle
- An Info.plist is a dictionary of key-value pairs
- Keys can be required or optional and are defined by CFBundle, CFPlugIn, or Finder
- Developers can add their own custom keys as well and easily access them at runtime
- The Info.plist is stored as a standard XML property list

Info.plist Keys
CFBundle

- Some of the standard keys supported by CFBundle include the following
  - CFBundleName
  - CFBundleExecutable
  - CFBundleVersion
  - CFBundleIdentifier
  - CFBundleDevelopmentRegion

Info.plist Keys
CFPlugIn

- Some of the standard keys supported by CFPlugIn include the following
  - CFPlugInDynamicRegistration
  - CFPlugInDynamicRegisterFunction
  - CFPlugInUnloadFunction
  - CFPlugInFactories
  - CFPlugInTypes

Resources—Files

- CFBundle provides API for locating file-based resources packaged with a bundle
- The API understands and encapsulates the internal structure of a bundle
- The API automatically deals with finding global or localized resources
- For localized resources, CFBundle chooses the language which is most preferred by the user among those available in the bundle

Resources—Forks

- Carbon will provide API based on CFBundle which allows localized versions of a file's resource fork to be used
- The correct localized resource fork for an application will be used automatically
- API will be provided for finding and using other localized resource forks
- Carbon apps will have the choice of using resource forks or using file based resources

Dynamic Code Loading

- CFBundle provides an abstract API for dynamically loading code
- This API insulates the developer from the actual binary format of the code
    - CFM, DYLD or DLL
- On Mac OS X where there are two supported binary formats, CFBundle will provide the ability to call across ABI boundaries

Summary

- CFPlugIn provides a new model applications can use to support plug-ins
- OS X and Carbon will support a new way to package your applications, frameworks, and plug-ins on disk
- CFBundle provides API for resource location and abstract code loading

Q&A

Roadmap

Carbon Overview:	Hall 2 **Tues., 1:00PM**
Carbon on Mac OS 8:	Hall A1 **Tues., 2:30pm**
Carbon on Mac OS X:	Hall A1 **Tues., 4:00pm**
Core Foundation Overview:	Hall 2 **Wed., 1:00pm**

Think different.™

Welcome

To Advance through Presentation
Use Page Up and Page Down Keys