Michael Hutchinson

Software Engineer

Xamarin

mhutch@xamarin.com

@mjhutchinson

http://mhut.ch

Extending Xamarin Studio with Addins

Xamarin

The Addin System

- The addin is the unit of packaging for extensibility
- Addin manifest (*.addin.xml) and/or assembly attributes declare:
  - Metadata and addin dependencies
  - Extensions and extension points
  - File list for packaging/distribution
- Extensions plug code and/or data into extension points, which may be in other addins

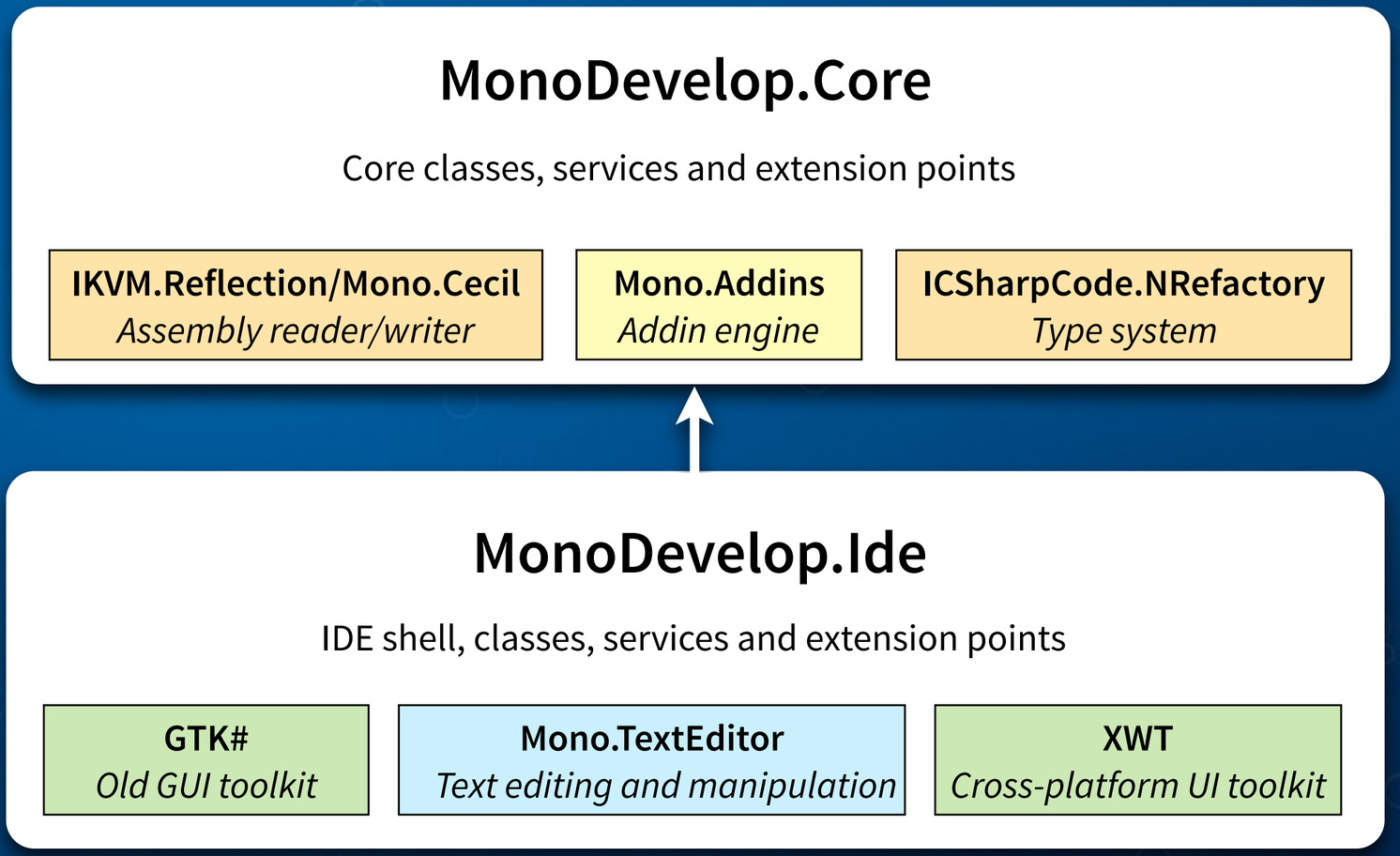
DEMO
Creating a Simple Addin

http://monodevelop.com/Developers/Articles/Creating_a_Simple_Add-in

https://github.com/mhutch/MonoDevelop.Samples.DateInserter

Core APIs

Core Addins


MonoDevelop.Core
Core classes, services and extension points
IKVM.Reflection/Mono.Cecil
Assembly reader/writer
Mono.Addins
Addin engine
ICSharpCode.NRefactory
Type system
MonoDevelop.Ide
IDE shell, classes, services and extension points
GTK#
Old GUI toolkit
Mono.TextEditor
Text editing and manipulation
XWT
Cross-platform UI toolkit

MonoDevelop.Core Services 1

- LoggingService
  - Log error/warning/info/debug messages to the IDE log

```
LoggingService.LogError ("Failed to download {0}: {1}", url, ex);
```

- PropertyService
  - Access and store global user settings

```
bool enabled = PropertyService.Get ("MyAddin.SettingName", false);
```

MonoDevelop.Core Services 2

- FileService
  - Global, batched dispatch of file events
  - More specific file events available on project model

```
FileService.NotifyFileChanged (someFile);
```

- StringParserService
  - Substitute values into placeholders in strings
  - Built-in contextual properties, or provide your own

```
var val = StringParserService.Parse ("Foo $(Prop)", properties);
```

MonoDevelop.Core Services 3

- Runtime.ProcessService
  - Run managed/native processes and read output

```
var args = new ProcessArgumentBuilder ();
args.AddQuotedFormat ("Hello \"{ 0}\"!", name);
var op = Runtime.ProcessService.StartProcess (
    "echo", args.ToString(), null, outWriter, errWriter, null);
op.WaitForOutput ();
```

- Runtime.SystemAssemblyService
  - Look up frameworks, runtimes and assemblies

```
var fx = Runtime.SystemAssemblyService.GetTargetFrameworkForAssembly (
    project.TargetRuntime,
    someAssemblyPath);
```

MonoDevelop.Core.FilePath

- Strongly typed string wrapper struct
- Makes it easy to use path manipulation functions
- Used almost everywhere we deal with file paths
- Comparisons respect OS case sensitivity
- Implicit cast to/from string

```
aPath.Combine ("foo").ToRelative (otherPath.ParentDirectory);
```

Progress And Status

- MonoDevelop.Core.IProgressMonitor
  - Report progress and errors, and check for cancellation
- MonoDevelop.Core.IAsyncOperation
  - Returned by methods as handle to async operations
- MonoDevelop.Core.ProgressMonitoring
  - Base classes, simple implementations and helpers
- Planned to be replaced by CancellationToken and Task<T>
  - Will allow using C# 5 async

Project Model

- Defined in MonoDevelop.Core.Projects
- Deserialized from native MSBuild format
- Solution contains SolutionItems
    - Projects and SolutionFolders
- .NET projects derived from DotNetProject
    - Subclass of Project
- Can define your own subclasses for specialized project types
    - Many virtual methods to customize behaviors
    - See e.g. AspNetProject

DotNetProject

- Files

```
ProjectFile file = dnp.Files.GetFileWithVirtualPath ("Foo.cfg");
file.BuildAction = BuildAction.None;
```

- Configurations

```
var debugSel = new SolutionConfigurationSelector ("Debug");
var cfg = (DotNetProjectConfiguration) dnp.GetConfiguration (debugSel);
var myProp = cfg.ExtendedProperties ["MyProperty"];
```

- Many useful properties, e.g. UserProperties, Policies, CompileTarget, FileName, BaseIntermediateOutputPath
- Many useful events, e.g. Saved, ConfigurationAdded, FileChangedInProject

IDE APIs

MonoDevelop.Ide Services

- IdeApp.Workbench
  - Root of loaded project model
- IdeApp.ProjectOperations
  - Build, execute, save, add files, etc.
- IdeApp.Workspace
  - Documents, pads, status bar, progress monitors
- IdeApp.Preferences
  - Strongly typed access to user preferences
- DesktopService
  - Abstraction over platform-specific operations

Type System

- MonoDevelop.Ide.TypeSystem.TypeSystemService
- File parsing

```
ParsedDocument doc = TypeSystemService.ParseFile (project, filePath);
var loc = new ICSharpCode.NRefactory.TextLocation (23, 5);
ICSharpCode.NRefactory.TypeSystem.IUnresolvedMember member =
    doc.GetMember (loc);
```

- Project/assembly type databases

```
var compilation = TypeSystemService.GetCompilation (project);
var typeName = new ICSharpCode.NRefactory.TypeSystem.TopLevelTypeName (
    "System.Core.Linq.Enumerable");
var typeDefinition = compilation.MainAssembly.GetTypeDefinition (typeName);
```

Common Extension Points

Defining and Exposing Commands

- Definition is separate from handling

```
<Extension path = "/MonoDevelop/Ide/Commands/Tools">
    <Command id = "MyAddin.MyAddinCommands.MyTool"
         _label = "Do _Stuff"
       shortcut = "Ctrl|D"
    description = "Does stuff"/>
</Extension>
```

- ID must be backed by an enum value
- Commands can be exposed via menu extension points

```
<Extension path = "/MonoDevelop/Ide/MainMenu/Tools">
    <CommandItem id = "MyAddin.MyAddinCommands.MyTool" />
</Extension>
```

Handling Commands

- If possible, handle existing command, don’t define new one
- Contextual, scans widget hierarchy for handler attributes
  - Extensible widgets direct the scan down extensions
  - Falls back to the optional defaultHandler on definition
  - If no handler found, command is disabled
- MonoDevelop.Components.Commands
  - CommandHandler
  - CommandHandlerAttribute
  - CommandUpdateHandlerAttribute

File and Project Templates

- Simple xft/xpt manifest with metadata and list of files
- Far too many properties and file processors to list here
  - See existing examples in MD codebase

```
<Extension path = "/MonoDevelop/Ide/FileTemplates">
    <FileTemplate id = "MyTemplate" file = "MyTemplate.xft.xml"/>
</Extension>
```

```
<Template>
    <TemplateConfiguration>
        <_Name>My Template</_Name>
        <_Category>Stuff</_Category>
        <_Description>An awesome template.</_Description>
    </TemplateConfiguration>
    <TemplateFiles>
        <File name="${Name}.cs" src="MyFile.cs" />
    </TemplateFiles>
</Template>
```

Document Editors

- Interfaces and classes in MonoDevelop.Ide.Gui
- IViewDisplayBinding is primary handler for a file
  - Returns AbstractViewContent
- IAttachableDisplayBinding provides alternate view of file
  - Return AbstractAttachableViewContent subclass
- Content can implement MonoDevelop.Ide.Gui.Content interfaces to opt into additional functionality
  - IPathedDocument, INavigable, IEditableTextBuffer, etc
- Also MonoDevelop.DesignerSupport interfaces
  - IPropertyPadProvider, IOutlinedDocument

Parsers

- Parsers parse documents into AST used by other things
  - Code folding, error underlines, outlines, completion, type database, semantic highlighting, etc.

```
class MyParser : MonoDevelop.Ide.TypeSystem.TypeSystemParser {
    public override MonoDevelop.Ide.TypeSystem.ParsedDocument Parse (
        bool storeAst, string fileName, TextReader content,
        Project project = null)
    {
        return new MyParsedDocument (ParseContent (content));
    }
}
```

```
<Extension path = "/MonoDevelop/TypeSystem/Parser">
    <Parser class = "MyAddin.MyParser" mimeType="my/mimetype" />
</Extension>
```

Text Editor Extensions

- MonoDevelop.Ide.Gui.Content.TextEditorExtension
- Horizontal extension hosted by source editor
  - Virtual methods for intercepting keystrokes
  - Can implement content interfaces
  - Can implement command handlers
- For code completion, subclass CompletionTextEditorExtension and override HandleCodeCompletion
  - Many other overrides for parameter tooltips, snippets, etc

Generators

- When file with Custom Tool saved, generator generates child file
- MonoDevelop.Ide.CustomTools.ISingleFileCustomTool

```
public class FooToBarGenerator : ISingleFileCustomTool
{
    public IAsyncOperation Generate (IProgressMonitor monitor,
        ProjectFile file, SingleFileCustomToolResult result)
    {
        var p = file.FilePath.ChangeExtension (".bar");
        result.GeneratedFilePath = p;
        var src = File.ReadAllText (file.FilePath);
        File.WriteAllText (p, src.Replace ("foo", "bar"));
    }
}
```

```
<Extension path = "/MonoDevelop/Ide/CustomTools">
    <Tool name="MyGenerator" type="MyAddin.FooToBarGenerator"/>
</Extension>
```

Project Types

- Generally subclass DotNetProject, Project or SolutionItem
- Implement MonoDevelop.Projects.IProjectBinding extension to support project creation
- If using configuration subclass, beware cloning
- Automatic property serialization via attributes
- Register type with flavor GUID and targets

```
<Extension path = "/MonoDevelop/ProjectModel/MSBuildItemTypes">
    <DotNetProjectSubtype type="MyAddin.MyProjectType"
                          guid="{973CB239-B201-44F3-9AF9-6C6584645B36}"
                      useXBuild="true">
        <AddImport projects=
            "$(MSBuildExtensionsPath)\MyTargets.targets" />
    </DotNetProjectSubtype>
</Extension>
```

Code Issues

- Code Issues inspect the AST and provide suggestions, hints, warnings and errors, and optionally provide automatic fixes
- Extension point in MonoDevelop.Refactoring

```
<Extension path = "/MonoDevelop/Refactoring/CodeIssues">
    <CodeIssue mimeType = "text/x-csharp"
                  class = "MyAddin.FindCodethatlooksLikeJava"
               severity = "Warning" />
</Extension>
```

- Many exist in NRefactory, with helpers for pattern matching, flow analysis, etc.
- http://mikemdblog.blogspot.com/2012/04/how-to-write-c-issue-provider.html

Context Actions

- Context Actions inspect AST and provide useful transformations
- Extension point in MonoDevelop.Refactoring

```
<Extension path = "/MonoDevelop/Refactoring/CodeActions">
    <Action mimeType = "text/x-csharp"
                class = "MyAddin.UnrollLoopAction"
               _title = "Unroll loop"
         _description = "Unrolls a for loop." />
</Extension>
```

- Implementation very similar to Code Issues
    - Main difference is the UX
- http://mikemdblog.blogspot.com/2012/03/how-to-write-context-action-using.html

```
[IssueDescription("Remove redundant 'internal' modifier",
       Description="Removes 'internal' modifiers that are not required.",
       Category = IssueCategories.Redundancies,
       Severity = Severity.Hint,
       IssueMarker = IssueMarker.GrayOut)]
public class RedundantInternalIssue : ICodeIssueProvider
{
   public IEnumerable<CodeIssue> GetIssues(BaseRefactoringContext context)
   {
      return new GatherVisitor(context, this).GetIssues();
   }

   class GatherVisitor : GatherVisitorBase<RedundantInternalIssue>
   {
      public GatherVisitor (BaseRefactoringContext ctx, RedundantInternalIssue issue) : base (ctx, issue)
      {
      }

      public override void VisitTypeDeclaration(TypeDeclaration typeDeclaration)
      {
         foreach (var token_ in typeDeclaration.ModifierTokens) {
            var token = token_;
            if (token.Modifier == Modifiers.Internal) {
               AddIssue(token, ctx.TranslateString("Remove 'internal' modifier"), script => {
                  int offset = script.GetCurrentOffset(token.StartLocation);
                  int endOffset = script.GetCurrentOffset(token.GetNextNode().StartLocation);
                  script.RemoveText(offset, endOffset - offset);
               });
            }
         }
      }
   }
}
```

Build Customization

- If possible, use MSBuild/xbuild targets for build/clean
  - New project types, existing MSBuild-based project types
  - Extremely powerful, can customize almost anything
  - Works with MSBuild/xbuild and Visual Studio
  - Doesn’t even require an addin
  - But see https://mhut.ch/journal/2012/08/19/state_msbuild_support_monodevelop
- Otherwise, use MonoDevelop.Projects.ProjectServiceExtension
  - Horizontal extension
  - Can intercept/handle build, clean, execute, load, etc

And Many, Many More

- Pads
- Key binding schemes
- Policies
- Code formatters
- Project file formats
- Preferences panels
- Options panels
- Debugger protocols
- Debugger visualizers
- Workspace layouts
- Solution pad tree nodes
- Source editor margins
- Unit test engines
- Code generators
- Code snippets
- Target frameworks
- Target runtime
- VCS backends
- Refactorings
- Execution handlers
- Syntax highlighting
- ...

DEMO
Tour of an Addin

MonoDevelop.TextTemplating

Publishing Addins

- Simple way is to pack the addin

```
mdtool pack youraddin.dll
```

- Resulting mpack file can be installed from the Addin Manager
  - Tools menu (Windows), Xamarin Studio menu (Mac)
- Or use the addin build server
  - http://addins.monodevelop.com
  - Publishes to online addin repository
  - Installable from Addin Gallery

Where Next?

- http://monodevelop.com/Developers/Articles
- irc://irc.gnome.org/#monodevelop
- monodevelop-devel-list@lists.ximian.com
- https://github.com/mono/monodevelop
- https://github.com/icsharpcode/NRefactory

Thank You

Questions?