```
// This file is part of Visual D
//
// Visual D integrates the D programming language into Visual Studio
// Copyright (c) 2010 by Rainer Schuetze, All Rights Reserved
//
// Distributed under the Boost Software License, Version 1.0.
// See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt

module visuald.dpackage;

import visuald.windows;
import core.stdc.stdlib;
import std.windows.charset;
import std.string;
import std.utf;
import std.path;
import std.file;
import std.conv;
import std.array;
import std.exception;
import std.algorithm;
static import std.ascii;

import stdext.path;
import stdext.array;
import stdext.file;
import stdext.string;
import stdext.registry;

import visuald.comutil;
import visuald.hierutil;
import visuald.stringutil;
import visuald.fileutil;
import visuald.dproject;
import visuald.automation;
import visuald.build;
```

```
import visuald.config;
import visuald.chiernode;
import visuald.dlangsvc;
import visuald.dimagelist;
import visuald.logutil;
import visuald.propertypage;
import visuald.winctrl;
import visuald.register;
import visuald.intellisense;
import visuald.searchsymbol;
import visuald.tokenreplacedialog;
import visuald.cppwizard;
import visuald.profiler;
import visuald.library;
import visuald.pkgutil;
import visuald.colorizer;
import visuald.dllmain;
import visuald.taskprovider;
import visuald.vdserverclient;
import xml = visuald.xmlwrap;

import sdk.win32.winreg;
import sdk.win32.oleauto;

import sdk.vsi.vsshell;
import sdk.vsi.vssplash;
import sdk.vsi.proffserv;
import sdk.vsi.vsshell90;
import sdk.vsi.objext;
static import dte = sdk.vsi.dte80a;
static import dte2 = sdk.vsi.dte80;

///////////////////////////////////////////////////////////////////////

struct LanguageProperty
{
    wstring name;
    DWORD value;
}
```

```
const string pkg_version  = extractDefine(import("version"), "VERSION_MAJOR") ~ "." ~
                        extractDefine(import("version"), "VERSION_MINOR");
const string plk_version  = "0.3"; // for VS2008 or earlier
const string bld_version  = extractDefine(import("version"), "VERSION_BUILD");
const string beta_version = extractDefine(import("version"), "VERSION_BETA");
const string full_version = pkg_version  ~ "." ~
                        extractDefine(import("version"), "VERSION_REVISION") ~
                        (bld_version != "0" ? beta_version ~ bld_version : "");

/*-------------------------------------------------------
 * Globals
 *-------------------------------------------------------*/
const wstring g_languageName            = "D"w;
const wstring g_packageName             = "Visual D"w;
const  string g_packageVersion          = pkg_version;
const wstring g_packageCompany          = "Rainer Schuetze"w;
const wstring[] g_languageFileExtensions = [ ".d"w, ".di"w, ".mixin"w ];
const wstring g_defaultProjectFileExtension = "visualdproj"w;
const wstring[] g_projectFileExtensions  = [ "visualdproj"w, "dproj"w ];

// CLSID registered in extensibility center (PLK)
const GUID    g_packageCLSID            = uuid("002a2de9-8bb6-484d-987f-7e4ad4084715");
const GUID    g_vendorCLSID             = uuid("002a2de9-8bb6-484d-987e-7e4ad4084715");

const GUID    g_languageCLSID           = uuid("002a2de9-8bb6-484d-9800-7e4ad4084715");
const GUID    g_projectFactoryCLSID     = uuid("002a2de9-8bb6-484d-9802-
7e4ad4084715");
const GUID    g_intellisenseCLSID       = uuid("002a2de9-8bb6-484d-9801-7e4ad4084715");
const GUID    g_commandSetCLSID         = uuid("002a2de9-8bb6-484d-9803-
7e4ad4084715");
const GUID    g_searchWinCLSID          = uuid("002a2de9-8bb6-484d-9804-7e4ad4084715");
const GUID    g_debuggerLanguage        = uuid("002a2de9-8bb6-484d-9805-
7e4ad4084715");
const GUID    g_expressionEvaluator     = uuid("002a2de9-8bb6-484d-9806-7e4ad4084715");
const GUID    g_profileWinCLSID         = uuid("002a2de9-8bb6-484d-9807-7e4ad4084715");
const GUID    g_tokenReplaceWinCLSID    = uuid("002a2de9-8bb6-484d-9808-
7e4ad4084715");
const GUID    g_outputPaneCLSID         = uuid("002a2de9-8bb6-484d-9809-
```

```
7e4ad4084715");
const GUID    g_CppWizardWinCLSID        = uuid("002a2de9-8bb6-484d-980a-
7e4ad4084715");

const GUID    g_omLibraryManagerCLSID    = uuid("002a2de9-8bb6-484d-980b-
7e4ad4084715");
const GUID    g_omLibraryCLSID           = uuid("002a2de9-8bb6-484d-980c-7e4ad4084715");
const GUID    g_ProjectItemWizardCLSID   = uuid("002a2de9-8bb6-484d-980d-
7e4ad4084715");

const GUID    g_unmarshalEnumOutCLSID    = uuid("002a2de9-8bb6-484d-980e-
7e4ad4084715");
// const GUID g_unmarshalTargetInfoCLSID = uuid("002a2de9-8bb6-484d-980f-
7e4ad4084715"); // defined in config.d

const GUID    g_VisualDHelperCLSID       = uuid("002a2de9-8bb6-484d-aa10-
7e4ad4084715");
const GUID    g_VisualCHelperCLSID       = uuid("002a2de9-8bb6-484d-aa11-
7e4ad4084715");

// more guids in propertypage.d starting with 9810

const LanguageProperty[] g_languageProperties =
[
   // see http://msdn.microsoft.com/en-us/library/bb166421.aspx
   { "RequestStockColors"w,          0 },
   { "ShowCompletion"w,              1 },
   { "ShowSmartIndent"w,             1 },
   { "ShowHotURLs"w,                 1 },
   { "Default to Non Hot URLs"w,     1 },
   { "DefaultToInsertSpaces"w,       0 },
   { "ShowDropdownBarOption "w,      1 },
   { "Single Code Window Only"w,     1 },
   { "EnableAdvancedMembersOption"w, 1 },
   { "Support CF_HTML"w,             0 },
   { "EnableLineNumbersOption"w,     1 },
   { "HideAdvancedMembersByDefault"w, 0 },
   { "ShowBraceCompletion"w,         1 },
```

```
];

//////////////////////////////////////////////////////////////////////////
void global_init()
{
    // avoid cyclic init dependencies
    initWinControls(g_hInst);
    LanguageService.shared_static_this();
    CHierNode.shared_static_this();
    CHierNode.shared_static_this_typeHolder();
    automation_shared_static_this_typeHolder();
    Project.shared_static_this_typeHolder();
}

void global_exit()
{
    LanguageService.shared_static_dtor();
    CHierNode.shared_static_dtor_typeHolder();
    automation_shared_static_dtor_typeHolder();
    Project.shared_static_dtor_typeHolder();
    Package.s_instance = null;
}

//////////////////////////////////////////////////////////////////////////
__gshared int g_dllRefCount;

extern(Windows)
HRESULT DllCanUnloadNow()
{
    return (g_dllRefCount == 0) ? S_OK : S_FALSE;
}

extern(Windows)
HRESULT DllGetClassObject(CLSID* rclsid, IID* riid, LPVOID* ppv)
{
    logCall("DllGetClassObject(rclsid=%s, riid=%s)", _toLog(rclsid), _toLog(riid));

    if(*rclsid == g_packageCLSID)
    {
```

```
        auto factory = newCom!ClassFactory;
        return factory.QueryInterface(riid, ppv);
    }
    if(*rclsid == g_unmarshalEnumOutCLSID)
    {
        DEnumOutFactory eof = newCom!DEnumOutFactory;
        return eof.QueryInterface(riid, ppv);
    }
    static if(is(typeof(g_unmarshalTargetInfoCLSID))) if(*rclsid ==
g_unmarshalTargetInfoCLSID)
    {
        TargetInfoFactory eof = newCom!TargetInfoFactory;
        return eof.QueryInterface(riid, ppv);
    }
    if(*rclsid == g_ProjectItemWizardCLSID)
    {
        auto wiz = newCom!WizardFactory;
        return wiz.QueryInterface(riid, ppv);
    }
    if(PropertyPageFactory factory = PropertyPageFactory.create(rclsid))
        return factory.QueryInterface(riid, ppv);

    return E_NOINTERFACE;
}

////////////////////////////////////////////////////////////////////////
class ClassFactory : DComObject, IClassFactory
{
    this() {}

    override HRESULT QueryInterface(in IID* riid, void** pvObject)
    {
        if(queryInterface2!(IClassFactory) (this, IID_IClassFactory, riid, pvObject))
            return S_OK;
        return super.QueryInterface(riid, pvObject);
    }

    override HRESULT CreateInstance(IUnknown UnkOuter, in IID* riid, void** pvObject)
    {
```

```
        logCall("%s.CreateInstance(riid=%s)", this, _toLog(riid));

        if(*riid == g_languageCLSID)
        {
            assert(!UnkOuter);
            LanguageService service = newCom!LanguageService(null);
            return service.QueryInterface(riid, pvObject);
        }
        if(*riid == IVsPackage.iid)
        {
            assert(!UnkOuter);
            Package pkg = newCom!Package;
            return pkg.QueryInterface(riid, pvObject);
        }
        if(*riid == g_unmarshalEnumOutCLSID)
        {
            assert(!UnkOuter);
            DEnumOutputs eo = newCom!DEnumOutputs(null, 0);
            return eo.QueryInterface(riid, pvObject);
        }
        static if(is(typeof(g_unmarshalTargetInfoCLSID))) if(*riid ==
g_unmarshalTargetInfoCLSID)
        {
            assert(!UnkOuter);
            auto pti = newCom!ProfilerTargetInfo(null);
            return pti.QueryInterface(riid, pvObject);
        }
        return S_FALSE;
    }

    override HRESULT LockServer(in BOOL fLock)
    {
        if(fLock)
            InterlockedIncrement(&g_dllRefCount);
        else
            InterlockedDecrement(&g_dllRefCount);
        return S_OK;
    }
```

```
    int lockCount;
}

//////////////////////////////////////////////////////////////////////

static const GUID SOleComponentManager_iid = { 0x000C060B,0x0000,0x0000,[
0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x46 ] };

//////////////////////////////////////////////////////////////////////
class Package : DisposingComObject,
        IVsPackage,
        IServiceProvider,
        IVsInstalledProduct,
        IOleCommandTarget,
        IOleComponent,
        IVsPersistSolutionProps // inherits IVsPersistSolutionOpts
{
    __gshared Package s_instance;

    this()
    {
        s_instance = this;
        mOptions = new GlobalOptions();
        mLangsvc = addref(newCom!LanguageService(this));
        mProjFactory = addref(newCom!ProjectFactory(this));
        mLibInfos = new LibraryInfos();
    }

    ~this()
    {
    }

    override HRESULT QueryInterface(in IID* riid, void** pvObject)
    {
        if(queryInterface!(IVsPackage) (this, riid, pvObject))
            return S_OK;
        if(queryInterface!(IServiceProvider) (this, riid, pvObject))
            return S_OK;
        if(queryInterface!(IVsInstalledProduct) (this, riid, pvObject))
```

```
            return S_OK;
        if(queryInterface!(IOleCommandTarget) (this, riid, pvObject))
            return S_OK;
        if(queryInterface!(IOleComponent) (this, riid, pvObject))
            return S_OK;
        if(queryInterface!(IVsPersistSolutionOpts) (this, riid, pvObject))
            return S_OK;
        if(queryInterface!(IVsPersistSolutionProps) (this, riid, pvObject))
            return S_OK;
        return super.QueryInterface(riid, pvObject);
    }

    override void Dispose()
    {
        deleteVisualDOutputPane();

        Close();
        mLangsvc = release(mLangsvc);
        mProjFactory = release(mProjFactory);
        if(s_instance == this)
            s_instance = null;
    }

    // IVsPackage
    override int Close()
    {
        mixin(LogCallMix);

        if(mHostSP)
        {
            CloseLibraryManager();

            if(mLangServiceCookie)
            {
                IProfferService sc;
                if(mHostSP.QueryService(&IProfferService.iid, &IProfferService.iid, cast(void**)&sc)
== S_OK)
                {
                    if(mLangServiceCookie && sc.RevokeService(mLangServiceCookie) != S_OK)
```

```
            {
                OutputDebugLog("RevokeService(lang-service) failed");
            }
            sc.Release();
        }
        mLangServiceCookie = 0;
        if(mLangsvc)
            mLangsvc.Dispose(); // cannot call later because Package.mHostSP needed to
query services
        mLangsvc = release(mLangsvc);
    }
    if(mProjFactoryCookie)
    {
        IVsRegisterProjectTypes projTypes;
        if(mHostSP.QueryService(&IVsRegisterProjectTypes.iid,
&IVsRegisterProjectTypes.iid, cast(void**)&projTypes) == S_OK)
        {
            if(projTypes.UnregisterProjectType(mProjFactoryCookie) != S_OK)
            {
                OutputDebugLog("UnregisterProjectType() failed");
            }
            projTypes.Release();
        }
        mProjFactoryCookie = 0;
        mProjFactory = release(mProjFactory);
    }
    if (mComponentID != 0)
    {
        IOleComponentManager componentManager;
        if(mHostSP.QueryService(&SOleComponentManager_iid,
&IOleComponentManager.iid, cast(void**)&componentManager) == S_OK)
        {
            scope(exit) release(componentManager);
            componentManager.FRevokeComponent(mComponentID);
            mComponentID = 0;
        }
    }
    if (mTaskProviderCookie != 0)
    {
```

```
            if (auto taskList = queryService!(IVsTaskList))
            {
                scope(exit) release(taskList);
                taskList.UnregisterTaskProvider(mTaskProviderCookie);
                mTaskProvider = null;
                mTaskProviderCookie = 0;
            }
        }

        mHostSP = release(mHostSP);
    }
    return S_OK;
}

override int CreateTool(in GUID* rguidPersistenceSlot)
{
    mixin(LogCallMix);
    return E_NOTIMPL;
}
override int GetAutomationObject(in wchar* pszPropName, IDispatch* ppDisp)
{
    mixin(LogCallMix);
    return E_NOTIMPL;
}

override int GetPropertyPage(in GUID* rguidPage, VSPROPSHEETPAGE* ppage)
{
    mixin(LogCallMix2);

    ResizablePropertyPage tpp;
    if(*rguidPage == g_DmdDirPropertyPage)
        tpp = newCom!DmdDirPropertyPage(mOptions);
    else if(*rguidPage == g_GdcDirPropertyPage)
        tpp = newCom!GdcDirPropertyPage(mOptions);
    else if(*rguidPage == g_LdcDirPropertyPage)
        tpp = newCom!LdcDirPropertyPage(mOptions);
    else if(*rguidPage == g_ToolsProperty2Page)
        tpp = newCom!ToolsProperty2Page(mOptions);
    else if(*rguidPage == g_ColorizerPropertyPage)
```

```
        tpp = newCom!ColorizerPropertyPage(mOptions);
    else if(*rguidPage == g_IntellisensePropertyPage)
        tpp = newCom!IntellisensePropertyPage(mOptions);
    else if(*rguidPage == g_MagoPropertyPage)
        tpp = newCom!MagoPropertyPage();
    else
        return E_NOTIMPL;

    PROPPAGEINFO pageInfo;
    pageInfo.cb = PROPPAGEINFO.sizeof;
    tpp.GetPageInfo(&pageInfo);
    *ppage = VSPROPSHEETPAGE.init;
    ppage.dwSize = VSPROPSHEETPAGE.sizeof;
    auto win = new PropertyWindow(null, WS_OVERLAPPED, "Visual D Settings", tpp);
    win.setRect(0, 0, pageInfo.size.cx, pageInfo.size.cy);
    ppage.hwndDlg = win.hwnd;

    RECT r;
    win.GetWindowRect(&r);
    tpp._Activate(win, &r, false);
    tpp.SetWindowSize(0, 0, pageInfo.size.cx, pageInfo.size.cy);
    addref(tpp);

    win.destroyDelegate = delegate void(Widget w)
    {
        if(auto o = tpp)
        {
            tpp = null;
            o.Deactivate();
            release(o);
        }
    };
    win.applyDelegate = delegate void(Widget w)
    {
        tpp.Apply();
    };
    return S_OK;
}
```

```
    override int QueryClose(int* pfCanClose)
    {
        mixin(LogCallMix2);
        *pfCanClose = 1;
        return S_OK;
    }
    override int ResetDefaults(in uint grfFlags)
    {
        mixin(LogCallMix);
        return E_NOTIMPL;
    }
    override int SetSite(IServiceProvider psp)
    {
        mixin(LogCallMix);

        mHostSP = release(mHostSP);
        mHostSP = addref(psp);

        IProfferService sc;
        if(mHostSP.QueryService(&IProfferService.iid, &IProfferService.iid, cast(void**)&sc) ==
S_OK)
        {
            if(sc.ProfferService(&g_languageCLSID, this, &mLangServiceCookie) != S_OK)
            {
                OutputDebugLog("ProfferService(language-service) failed");
            }
            sc.Release();
        }
version(none)
{
    // getting the debugger here causes crashes when installing/uninstalling other plugins
    //  command line used by installer: devenv /setup /NoSetupVSTemplates
        IVsDebugger debugger;
        if(mHostSP.QueryService(&IVsDebugger.iid, &IVsDebugger.iid, cast(void**)&debugger)
== S_OK)
        {
            mLangsvc.setDebugger(debugger);
            debugger.Release();
        }
```

```
}
    IVsRegisterProjectTypes projTypes;
    if(mHostSP.QueryService(&IVsRegisterProjectTypes.iid, &IVsRegisterProjectTypes.iid,
cast(void**)&projTypes) == S_OK)
    {
        if(projTypes.RegisterProjectType(&g_projectFactoryCLSID, mProjFactory,
&mProjFactoryCookie) != S_OK)
        {
            OutputDebugLog("RegisterProjectType() failed");
        }
        projTypes.Release();
    }

    mOptions.initFromRegistry();

    //register with ComponentManager for Idle processing
    IOleComponentManager componentManager;
    if(mHostSP.QueryService(&SOleComponentManager_iid, &IOleComponentManager.iid,
cast(void**)&componentManager) == S_OK)
    {
        scope(exit) release(componentManager);
        if (mComponentID == 0)
        {
            OLECRINFO crinfo;
            crinfo.cbSize = crinfo.sizeof;
            crinfo.grfcrf = olecrfNeedPeriodicIdleTime | olecrfNeedAllActiveNotifs |
olecrfNeedSpecActiveNotifs;
            crinfo.grfcadvf = olecadvfModal | olecadvfRedrawOff | olecadvfWarningsOff;
            crinfo.uIdleTimeInterval = 500;
            if(!componentManager.FRegisterComponent(this, &crinfo, &mComponentID))
                OutputDebugLog("FRegisterComponent failed");
        }
    }
    InitLibraryManager();

    if (auto taskList = queryService!(IVsTaskList))
    {
        scope(exit) release(taskList);
        mTaskProvider = newCom!TaskProvider;
```

```
        if (taskList.RegisterTaskProvider (mTaskProvider, &mTaskProviderCookie) != S_OK)
            mTaskProvider = null;
        else
            taskList.RefreshTasks(mTaskProviderCookie);
    }

    return S_OK; // E_NOTIMPL;
}

// IServiceProvider
override int QueryService(in GUID* guidService, in IID* riid, void ** ppvObject)
{
    mixin(LogCallMix);

    if(mLangsvc && *guidService == g_languageCLSID)
        return mLangsvc.QueryInterface(riid, ppvObject);
    if(mProjFactory && *guidService == g_projectFactoryCLSID)
        return mProjFactory.QueryInterface(riid, ppvObject);

    return E_NOTIMPL;
}

// IVsInstalledProduct
override int get_IdBmpSplash(uint* pIdBmp)
{
    mixin(LogCallMix);
    *pIdBmp = BMP_SPLASHSCRN;
    return S_OK;
}

override int get_OfficialName(BSTR* pbstrName)
{
    logCall("%s.ProductID(pbstrName=%s)", this, pbstrName);
    *pbstrName = allocwBSTR(g_packageName);
    return S_OK;
}
override int get_ProductID(BSTR* pbstrPID)
{
    logCall("%s.ProductID(pbstrPID=%s)", this, pbstrPID);
```

```
        *pbstrPID = allocBSTR(full_version);
        return S_OK;
    }
    override int get_ProductDetails(BSTR* pbstrProductDetails)
    {
        logCall("%s.ProductDetails(pbstrPID=%s)", this, pbstrProductDetails);
        *pbstrProductDetails = allocBSTR ("Integration of the D Programming Language into
Visual Studio");
        return S_OK;
    }

    override int get_IdIcoLogoForAboutbox(uint* pIdIco)
    {
        logCall("%s.IdIcoLogoForAboutbox(pIdIco=%s)", this, pIdIco);
        *pIdIco = ICON_ABOUTBOX;
        return S_OK;
    }

    // IOleCommandTarget ///////////////////////////////////
    override int QueryStatus(in GUID *pguidCmdGroup, in uint cCmds,
                             OLECMD *prgCmds, OLECMDTEXT *pCmdText)
    {
        mixin(LogCallMix);

        for (uint i = 0; i < cCmds; i++)
        {
            if(g_commandSetCLSID == *pguidCmdGroup)
            {
                switch(prgCmds[i].cmdID)
                {
                case CmdSearchFile:
                case CmdSearchSymbol:
                case CmdSearchTokNext:
                case CmdSearchTokPrev:
                case CmdReplaceTokens:
                case CmdConvWizard:
                case CmdDustMite:
                case CmdBuildPhobos:
                case CmdShowProfile:
```

```
            case CmdShowLangPage:
            case CmdShowWebsite:
            case CmdDelLstFiles:
               prgCmds[i].cmdf = OLECMDF_SUPPORTED | OLECMDF_ENABLED;
               break;
            default:
               prgCmds[i].cmdf = OLECMDF_SUPPORTED;
               break;
            }
         }
      }
      return S_OK;
}

override int Exec( /* [unique][in] */ in GUID *pguidCmdGroup,
         /* [in] */ in uint nCmdID,
         /* [in] */ in uint nCmdexecopt,
         /* [unique][in] */ in VARIANT *pvaIn,
         /* [unique][out][in] */ VARIANT *pvaOut)
{
   if(g_commandSetCLSID != *pguidCmdGroup)
      return OLECMDERR_E_NOTSUPPORTED;

   if(nCmdID == CmdSearchSymbol)
   {
      showSearchWindow(false);
      return S_OK;
   }
   if(nCmdID == CmdSearchFile)
   {
      showSearchWindow(true);
      return S_OK;
   }
   if(nCmdID == CmdSearchTokNext)
   {
      findNextTokenReplace(false);
      return S_OK;
   }
   if(nCmdID == CmdSearchTokPrev)
```

```
    {
        findNextTokenReplace(true);
        return S_OK;
    }
    if(nCmdID == CmdReplaceTokens)
    {
        showTokenReplaceWindow(true);
        return S_OK;
    }
    if(nCmdID == CmdConvWizard)
    {
        showCppWizardWindow();
        return S_OK;
    }
    if(nCmdID == CmdBuildPhobos)
    {
        mOptions.buildPhobosBrowseInfo();
        mLibInfos.updateDefinitions();
        return S_OK;
    }
    if(nCmdID == CmdDustMite)
    {
        return DustMiteProject();
    }
    if(nCmdID == CmdShowProfile)
    {
        showProfilerWindow();
        return S_OK;
    }
    if(nCmdID == CmdShowLangPage)
    {
        auto pIVsUIShell = ComPtr!(IVsUIShell)(queryService!(IVsUIShell), false);
        GUID targetGUID = uuid("734A5DE2-DEBA-11d0-A6D0-00C04FB67F6A");
        VARIANT var;
        var.vt = VT_BSTR;
        var.bstrVal = allocBSTR("002A2DE9-8BB6-484D-9823-7E4AD4084715");
        pIVsUIShell.PostExecCommand(&CMDSETID_StandardCommandSet97,
cmdidToolsOptions, OLECMDEXECOPT_DODEFAULT, &var);
        freeBSTR(var.bstrVal);
```

```
            return S_OK;
        }
        if(nCmdID == CmdShowWebsite)
        {
            if(dte2.DTE2 spvsDTE = GetDTE())
            {
                scope(exit) release(spvsDTE);
                spvsDTE.ExecuteCommand("View.WebBrowser"w.ptr,
"http://rainers.github.io/visuald/visuald/StartPage.html"w.ptr);
            }
            return S_OK;
        }
        if(nCmdID == CmdDelLstFiles)
        {
            GetGlobalOptions().DeleteCoverageFiles();
            return S_OK;
        }
        return OLECMDERR_E_NOTSUPPORTED;
    }

    // IOleComponent Methods
    BOOL FDoIdle(in OLEIDLEF grfidlef)
    {
        if(mWantsUpdateLibInfos)
        {
            mWantsUpdateLibInfos = false;
            Package.GetLibInfos().updateDefinitions();
        }
        OutputPaneBuffer.flush();

        if (mLangsvc.OnIdle())
            return true;

        return false;
    }

    void Terminate()
    {
    }
```

```
   BOOL FPreTranslateMessage(MSG* msg)
   {
      return FALSE;
   }
   void OnEnterState(in OLECSTATE uStateID, in BOOL fEnter)
   {
   }
   void OnAppActivate(in BOOL fActive, in DWORD dwOtherThreadID)
   {
   }
   void OnLoseActivation()
   {
   }
   void OnActivationChange(/+[in]+/ IOleComponent pic,
                           in BOOL fSameComponent,
                           in const( OLECRINFO)*pcrinfo,
                           in BOOL fHostIsActivating,
                           in const( OLECHOSTINFO)*pchostinfo,
                           in DWORD dwReserved)
   {
   }
   BOOL FReserved1(in DWORD dwReserved, in UINT message, in WPARAM wParam, in
LPARAM lParam)
   {
      return TRUE;
   }

   BOOL FContinueMessageLoop(in OLELOOP uReason, in void *pvLoopData, in MSG
*pMsgPeeked)
   {
      return 1;
   }
   BOOL FQueryTerminate( in BOOL fPromptUser)
   {
      return 1;
   }
   HWND HwndGetWindow(in OLECWINDOW dwWhich, in DWORD dwReserved)
   {
      return null;
```

```
    }

    ////////////////////////////////////////////////////////
    // IVsPersistSolutionOpts (writes to suo file)

    enum slnPersistenceOpts  = "VisualDProjectSolutionOptions"w;

    HRESULT SaveUserOptions(IVsSolutionPersistence pPersistence)
    {
        mixin(LogCallMix);
        return pPersistence.SavePackageUserOpts(this, slnPersistenceOpts.ptr);
    }
    HRESULT LoadUserOptions(IVsSolutionPersistence pPersistence, in VSLOADUSEROPTS
grfLoadOpts)
    {
        mixin(LogCallMix);
        return pPersistence.LoadPackageUserOpts(this, slnPersistenceOpts.ptr);
    }

    ////////////////////////////
    static HRESULT writeUint(IStream pStream, uint num)
    {
        ULONG written;
        HRESULT hr = pStream.Write(&num, num.sizeof, &written);
        if(hr == S_OK && written != num.sizeof)
            hr = E_FAIL;
        return hr;
    }
    static HRESULT writeGUID(IStream pStream, ref const GUID uid)
    {
        ULONG written;
        HRESULT hr = pStream.Write(&uid, uid.sizeof, &written);
        if(hr == S_OK && written != uid.sizeof)
            hr = E_FAIL;
        return hr;
    }
    static HRESULT writeString(IStream pStream, string s)
    {
        if(HRESULT hr = writeUint(pStream, cast(uint) s.length))
```

```
        return hr;

    ULONG written;
    HRESULT hr = pStream.Write(s.ptr, s.length, &written);
    if(hr == S_OK && written != s.length)
        hr = E_FAIL;
    return hr;
}
static HRESULT writeConfig(IStream pStream, Config cfg)
{
    if(auto hr = writeString(pStream, cfg.getName()))
        return hr;
    if(auto hr = writeString(pStream, cfg.getPlatform()))
        return hr;

    xml.Document doc = xml.newDocument("SolutionOptions");
    cfg.GetProjectOptions().writeDebuggerXML(doc);
    string[] result = xml.writeDocument(doc);
    string res = std.string.join(result, "\n");
    if(auto hr = writeString(pStream, res))
        return hr;

    return S_OK;
}
////////////////////////////
static HRESULT readRaw(IStream pStream, void* p, uint size)
{
    ULONG read;
    HRESULT hr = pStream.Read(p, size, &read);
    if(hr == S_OK && read != size)
        hr = E_FAIL;
    return hr;
}
static HRESULT readUint(IStream pStream, ref uint num)
{
    return readRaw(pStream, &num, num.sizeof);
}
static HRESULT readGUID(IStream pStream, ref GUID uid)
{
```

```
        return readRaw(pStream, &uid, uid.sizeof);
    }
    static HRESULT readString(IStream pStream, ref string s)
    {
        uint len;
        if(HRESULT hr = readUint(pStream, len))
            return hr;

        if(len == -1)
            return S_FALSE;
        char[] buf = new char[len];
        HRESULT hr = readRaw(pStream, buf.ptr, len);
        s = assumeUnique(buf);
        return hr;
    }
    static HRESULT skip(IStream pStream, uint len)
    {
        char[256] buf;
        for(; len >= buf.sizeof; len -= buf.sizeof)
            if(auto hr = readRaw(pStream, buf.ptr, buf.sizeof))
                return hr;

        if(len > 0)
            if(auto hr = readRaw(pStream, buf.ptr, len))
                return hr;
        return S_OK;
    }

    HRESULT WriteUserOptions(IStream pOptionsStream, in LPCOLESTR pszKey)
    {
        mixin(LogCallMix);

        auto srpSolution = queryService!(IVsSolution);
        if(srpSolution)
        {
            scope(exit) release(srpSolution);
            IEnumHierarchies pEnum;
            if(srpSolution.GetProjectEnum(EPF_LOADEDINSOLUTION|EPF_MATCHTYPE,
&g_projectFactoryCLSID, &pEnum) == S_OK)
```

```
        {
            scope(exit) release(pEnum);
            IVsHierarchy pHierarchy;
            while(pEnum.Next(1, &pHierarchy, null) == S_OK)
            {
                scope(exit) release(pHierarchy);
                if(IVsGetCfgProvider getCfgProvider = qi_cast!IVsGetCfgProvider(pHierarchy))
                {
                    scope(exit) release(getCfgProvider);
                    IVsCfgProvider cfgProvider;
                    if(getCfgProvider.GetCfgProvider(&cfgProvider) == S_OK)
                    {
                        scope(exit) release(cfgProvider);

                        GUID uid;
                        pHierarchy.GetGuidProperty(VSITEMID_ROOT,
VSHPROPID_ProjectIDGuid, &uid);
                        if(auto hr = writeGUID(pOptionsStream, uid))
                            return hr;

                        ULONG cnt;
                        if(cfgProvider.GetCfgs(0, null, &cnt, null) == S_OK)
                        {
                            IVsCfg[] cfgs = new IVsCfg[cnt];
                            scope(exit) foreach(c; cfgs) release(c);
                            if(cfgProvider.GetCfgs(cnt, cfgs.ptr, &cnt, null) == S_OK)
                            {
                                foreach(c; cfgs)
                                {
                                    if(Config cfg = qi_cast!Config(c))
                                    {
                                        scope(exit) release(cfg);
                                        if(auto hr = writeConfig(pOptionsStream, cfg))
                                            return hr;
                                    }
                                }
                            }
                        }
                        // length -1 as end marker
```

```
                    if(auto hr = writeUint(pOptionsStream, -1))
                        return hr;
                }
            }
        }
        GUID uid; // empty GUID as end marker of projects
        if(auto hr = writeGUID(pOptionsStream, uid))
            return hr;

        version(writeSearchPaneState)
        {
            // now followed by more chunks with (iid,length) heaser
            if(auto win = getSearchPane(false))
            {
                if(auto hr = writeGUID(pOptionsStream, SearchPane.iid))
                    return hr;
                if(HRESULT hr = win.SaveViewState(pOptionsStream))
                    return hr;
            }
            // empty GUID as end marker
            if(auto hr = writeGUID(pOptionsStream, uid))
                return hr;
        }
    }
    return S_OK;
}

HRESULT ReadUserOptions(IStream pOptionsStream, in LPCOLESTR pszKey)
{
     mixin(LogCallMix);
    auto srpSolution = queryService!(IVsSolution);
    if(!srpSolution)
        return E_FAIL;
    scope(exit) release(srpSolution);

    GUID uid;
    for(;;)
    {
```

```
if(auto hr = readGUID(pOptionsStream, uid))
    return hr;
if(uid == GUID_NULL)
    break;

IVsHierarchy pHierarchy;
if (HRESULT hr = srpSolution.GetProjectOfGuid(&uid, &pHierarchy))
    return hr;

scope(exit) release(pHierarchy);
IVsGetCfgProvider getCfgProvider = qi_cast!IVsGetCfgProvider(pHierarchy);
if (!getCfgProvider)
    return E_FAIL;
scope(exit) release(getCfgProvider);

IVsCfgProvider cfgProvider;
if(auto hr = getCfgProvider.GetCfgProvider(&cfgProvider))
    return hr;
scope(exit) release(cfgProvider);

IVsCfgProvider2 cfgProvider2 = qi_cast!IVsCfgProvider2(cfgProvider);
if(!cfgProvider2)
    return E_FAIL;
scope(exit) release(cfgProvider2);

for(;;)
{
    string name, platform, xmltext;
    HRESULT hrName = readString(pOptionsStream, name);
    if(hrName == S_FALSE)
        break;
    if(hrName != S_OK)
        return hrName;
    if (auto hr = readString(pOptionsStream, platform))
        return hr;
    if (auto hr = readString(pOptionsStream, xmltext))
        return hr;

    IVsCfg pCfg;
```

```
            if (cfgProvider2.GetCfgOfName(_toUTF16z(name), _toUTF16z(platform), &pCfg) ==
S_OK)
            {
                scope(exit) release(pCfg);
                if(Config cfg = qi_cast!Config(pCfg))
                {
                    scope(exit) release(cfg);
                    try
                    {
                        xmltext = `<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>` ~
xmltext;
                        xml.Document doc = xml.readDocument(xmltext);
                        cfg.GetProjectOptions().parseXML(doc);
                    }
                    catch(Exception e)
                    {
                        writeToBuildOutputPane(e.toString());
                        logCall(e.toString());
                    }
                }
            }
        }

        version(writeSearchPaneState)
        while(readGUID(pOptionsStream, uid) == S_OK)
        {
            if(uid == GUID_NULL)
                break;
            if (uid == SearchPane.iid)
            {
                if(auto win = getSearchPane(true))
                {
                    if(HRESULT hr = win.LoadViewState(pOptionsStream))
                        return hr;
                    continue;
                }
            }
            // skip chunk
```

```
            uint len;
            if(HRESULT hr = readUint(pOptionsStream, len))
                return hr;
            if(HRESULT hr = skip(pOptionsStream, len))
              return hr;
        }
        return S_OK;
    }

    ////////////////////////////////////////////////////////
    // IVsPersistSolutionProps (writes to sln file)

    enum slnPersistenceKey   = "VisualDProjectSolutionProperties"w;
    enum slnPersistenceValue = "TestValue"w;

    override HRESULT QuerySaveSolutionProps(IVsHierarchy pHierarchy,
VSQUERYSAVESLNPROPS *pqsspSave)
    {
        mixin(LogCallMix);
        Project prj = qi_cast!Project(pHierarchy);
        if(!prj)
            return E_NOINTERFACE;
        release(prj);
        *pqsspSave = QSP_HasNoProps;
        return S_OK;
    }
    override HRESULT SaveSolutionProps(IVsHierarchy pHierarchy, IVsSolutionPersistence
pPersistence)
    {
        mixin(LogCallMix);
        return pPersistence.SavePackageSolutionProps(false, pHierarchy, this,
slnPersistenceKey.ptr);
    }
    override HRESULT WriteSolutionProps(IVsHierarchy pHierarchy, in LPCOLESTR pszKey,
IPropertyBag pPropBag)
    {
        mixin(LogCallMix);
        Project prj = qi_cast!Project(pHierarchy);
        if(!prj)
```

```
            return E_NOINTERFACE;
        release(prj);

        version(none)
        {
            VARIANT var;
            var.vt = VT_BSTR;
            var.bstrVal = allocBSTR("Test");
            HRESULT hr = pPropBag.Write(slnPersistenceValue.ptr, &var);
            freeBSTR(var.bstrVal);
        }
        return S_OK;
    }
    override HRESULT ReadSolutionProps(IVsHierarchy pHierarchy, in LPCOLESTR
pszProjectName,
                                       in LPCOLESTR pszProjectMk, in LPCOLESTR pszKey,
                                       in BOOL fPreLoad, /+[in]+/ IPropertyBag pPropBag)
    {
        mixin(LogCallMix);
        if(slnPersistenceKey == to_wstring(pszKey))
        {
            VARIANT var;
            if(pPropBag.Read(slnPersistenceValue.ptr, &var, null) == S_OK)
            {
                if (var.vt == VT_BSTR)
                {
                    string value = detachBSTR(var.bstrVal);
                }
            }
        }
        return S_OK;
    }
    override HRESULT OnProjectLoadFailure(IVsHierarchy pStubHierarchy, in LPCOLESTR
pszProjectName,
                                          in LPCOLESTR pszProjectMk, in LPCOLESTR pszKey)
    {
        mixin(LogCallMix);
        return S_OK;
    }
```

```
////////////////////////////////////////////////////////////
HRESULT InitLibraryManager()
{
    if (mOmLibraryCookie != 0) // already init-ed
        return E_UNEXPECTED;

    HRESULT hr = E_FAIL;
    if(auto om = queryService!(IVsObjectManager, IVsObjectManager2))
    {
        scope(exit) release(om);

        mLibrary = newCom!Library;
        hr = om.RegisterSimpleLibrary(mLibrary, &mOmLibraryCookie);
        if(SUCCEEDED(hr))
            mLibrary.Initialize();
    }
    return hr;
}

HRESULT CloseLibraryManager()
{
    if (mOmLibraryCookie == 0) // already closed or not init-ed
        return S_OK;

    HRESULT hr = E_FAIL;
    if(auto om = queryService!(IVsObjectManager, IVsObjectManager2))
    {
        scope(exit) release(om);
        hr = om.UnregisterLibrary(mOmLibraryCookie);
        mLibrary.Close(); // attaches itself to SolutionEvents, so we need to break circular
reference
        mLibrary = null;
    }
    mOmLibraryCookie = 0;
    return hr;
}

////////////////////////////////////////////////////////////
```

```
IServiceProvider getServiceProvider()
{
    return mHostSP;
}

static LanguageService GetLanguageService()
{
    assert(s_instance);
    return s_instance.mLangsvc;
}

static ProjectFactory GetProjectFactory()
{
    assert(s_instance);
    return s_instance.mProjFactory;
}

static TaskProvider GetTaskProvider()
{
    assert(s_instance);
    return s_instance.mTaskProvider;
}

static GlobalOptions GetGlobalOptions()
{
    assert(s_instance);
    return s_instance.mOptions;
}

static LibraryInfos GetLibInfos()
{
    assert(s_instance);
    return s_instance.mLibInfos;
}

static Library GetLibrary()
{
    assert(s_instance);
    return s_instance.mLibrary;
```

```
    }

    static void scheduleUpdateLibrary()
    {
        assert(s_instance);
        s_instance.mWantsUpdateLibInfos = true;
    }

    static void RefreshTaskList()
    {
        if (auto taskList = queryService!(IVsTaskList))
        {
            taskList.RefreshTasks(s_instance.mTaskProviderCookie);
            release(taskList);
        }
    }

private:
    IServiceProvider mHostSP;
    uint             mLangServiceCookie;
    uint             mProjFactoryCookie;

    uint             mComponentID;

    LanguageService  mLangsvc;
    ProjectFactory   mProjFactory;

    TaskProvider     mTaskProvider;
    uint             mTaskProviderCookie;

    uint             mOmLibraryCookie;

    GlobalOptions    mOptions;
    LibraryInfos     mLibInfos;
    bool             mWantsUpdateLibInfos;
    Library          mLibrary;
}

struct CompilerDirectories
```

```
{
    string InstallDir;
    string ExeSearchPath;
    string ImpSearchPath;
    string LibSearchPath;
    string DisasmCommand;

    string ExeSearchPath64;
    string LibSearchPath64;
    bool   overrideIni64;
    string overrideLinker64;
    string overrideOptions64;
    string DisasmCommand64;

    string ExeSearchPath32coff;
    string LibSearchPath32coff;
    bool   overrideIni32coff;
    string overrideLinker32coff;
    string overrideOptions32coff;
    string DisasmCommand32coff;
}

enum enableShowMemUsage = false;

class GlobalOptions
{
    HKEY hConfigKey;
    HKEY hUserKey;
    wstring regConfigRoot;
    wstring regUserRoot;

    CompilerDirectories DMD;
    CompilerDirectories GDC;
    CompilerDirectories LDC;
    string IncSearchPath;
    string JSNSearchPath;

    string UserTypesSpec;
    int[wstring] UserTypes;
```

```
// evaluated once at startup
string WindowsSdkDir;
string UCRTSdkDir;
string UCRTVersion;
string DevEnvDir;
string VSInstallDir;
string VCInstallDir;
string VCToolsInstallDir; // used by VS 2017
string VisualDInstallDir;

string excludeFileDeps; // files/paths to exclude from dependency monitoring
bool timeBuilds;
bool elasticSpace = true;
bool sortProjects = true;
bool stopSolutionBuild;
bool showUptodateFailure;
bool demangleError = true;
bool optlinkDeps = true;
bool showMemUsage = false;
bool autoOutlining;
byte deleteFiles;  // 0: ask, -1: don't delete, 1: delete (obsolete)
bool parseSource;
bool pasteIndent;
bool expandFromSemantics;
bool expandFromBuffer;
bool expandFromJSON;
byte expandTrigger;
bool showTypeInTooltip;
bool semanticGotoDef;
bool useDParser;
bool mixinAnalysis;
bool UFCSExpansions;
byte sortExpMode;  // 0: alphabetically, 1: by type, 2: by declaration and scope
bool exactExpMatch;
string VDServerIID;
string compileAndRunOpts;
string compileAndDbgOpts;
int compileAndDbgEngine;
```

```
    string[] coverageBuildDirs;
    string[] coverageExecutionDirs;

    bool showCoverageMargin;
    bool ColorizeCoverage = true;
    bool ColorizeVersions = true;
    bool lastColorizeCoverage;
    bool lastColorizeVersions;
    bool lastUseDParser;

    this()
    {
    }

    bool getRegistryRoot()
    {
        if(hConfigKey)
            return true;

        BSTR bstrRoot;
        ILocalRegistry4 registry4 = queryService!(ILocalRegistry, ILocalRegistry4);
        if(registry4)
        {
            scope(exit) release(registry4);
            if(registry4.GetLocalRegistryRootEx(RegType_Configuration, cast(uint*)&hConfigKey,
&bstrRoot) == S_OK)
            {
                regConfigRoot = wdetachBSTR(bstrRoot);
                if(registry4.GetLocalRegistryRootEx(RegType_UserSettings, cast(uint*)&hUserKey,
&bstrRoot) == S_OK)
                    regUserRoot = wdetachBSTR(bstrRoot);
                else
                {
                    regUserRoot = regConfigRoot;
                    hUserKey = HKEY_CURRENT_USER;
                }
                return true;
            }
```

```
        }
        ILocalRegistry2 registry = queryService!(ILocalRegistry, ILocalRegistry2);
        if(registry)
        {
            scope(exit) release(registry);
            if(registry.GetLocalRegistryRoot(&bstrRoot) == S_OK)
            {
                regConfigRoot = wdetachBSTR(bstrRoot);
                hConfigKey = HKEY_LOCAL_MACHINE;

                regUserRoot = regConfigRoot;
                hUserKey = HKEY_CURRENT_USER;
                return true;
            }
        }
        return false;
    }

    void detectWindowsSDKDir()
    {
        // todo: detect Win10 SDK
        if(WindowsSdkDir.empty)
        {
            scope RegKey keySdk = new RegKey(HKEY_LOCAL_MACHINE,
"SOFTWARE\\Microsoft\\Microsoft SDKs\\Windows\\v8.1"w, false);
            WindowsSdkDir = toUTF8(keySdk.GetString("InstallationFolder"));
            if(!std.file.exists(buildPath(WindowsSdkDir, "Lib")))
                WindowsSdkDir = "";
        }
        if(WindowsSdkDir.empty)
        {
            scope RegKey keySdk = new RegKey(HKEY_LOCAL_MACHINE,
"SOFTWARE\\Microsoft\\Microsoft SDKs\\Windows\\v8.0"w, false);
            WindowsSdkDir = toUTF8(keySdk.GetString("InstallationFolder"));
            if(!std.file.exists(buildPath(WindowsSdkDir, "Lib")))
                WindowsSdkDir = "";
        }
        if(WindowsSdkDir.empty)
        {
```

```
            scope RegKey keySdk = new RegKey(HKEY_LOCAL_MACHINE,
"SOFTWARE\\Microsoft\\Microsoft SDKs\\Windows"w, false);
            WindowsSdkDir = toUTF8(keySdk.GetString("CurrentInstallFolder"));
            if(!std.file.exists(buildPath(WindowsSdkDir, "Lib")))
                WindowsSdkDir = "";
        }
        if(WindowsSdkDir.empty)
            if(char* psdk = getenv("WindowsSdkDir"))
                WindowsSdkDir = fromMBSz(cast(immutable)psdk);
        if(!WindowsSdkDir.empty)
            WindowsSdkDir = normalizeDir(WindowsSdkDir);
    }

    void detectUCRT()
    {
        if(UCRTSdkDir.empty)
        {
            if(char* psdk = getenv("UniversalCRTSdkDir"))
                UCRTSdkDir = normalizeDir(fromMBSz(cast(immutable)psdk));
            else
            {
                scope RegKey keySdk = new RegKey(HKEY_LOCAL_MACHINE,
"SOFTWARE\\Microsoft\\Windows Kits\\Installed Roots"w, false);
                UCRTSdkDir = normalizeDir(toUTF8(keySdk.GetString("KitsRoot10")));
            }
        }
        if(UCRTVersion.empty)
        {
            if(char* pver = getenv("UCRTVersion"))
                UCRTVersion = fromMBSz(cast(immutable)pver);
            else if(!UCRTSdkDir.empty)
            {
                string rootsDir = normalizeDir(UCRTSdkDir) ~ "Lib\\";
                try
                {
                    foreach(string f; dirEntries(rootsDir, "*", SpanMode.shallow, false))
                        if(std.file.isDir(f) && f > UCRTVersion)
                        {
                            string bname = baseName(f);
```

```
                    if(!bname.empty && isDigit(bname[0]))
                        UCRTVersion = bname;
                }
            }
            catch(Exception)
            {
            }
        }
    }

    void detectVSInstallDir()
    {
        if(char* pe = getenv("VSINSTALLDIR"))
            VSInstallDir = fromMBSz(cast(immutable)pe);
        else
        {
            scope RegKey keyVS = new RegKey(hConfigKey, regConfigRoot, false);
            VSInstallDir = toUTF8(keyVS.GetString("InstallDir"));
            // InstallDir is ../Common7/IDE/
            VSInstallDir = normalizeDir(VSInstallDir);
            VSInstallDir = dirName(dirName(VSInstallDir));
        }
        VSInstallDir = normalizeDir(VSInstallDir);
    }

    void detectVCInstallDir()
    {
        string defverFile = VSInstallDir ~
r"VC\Auxiliary\Build\Microsoft.VCToolsVersion.default.txt";
        if (std.file.exists(defverFile))
        {
            // VS 2017
            try
            {
                string ver = strip(readUtf8(defverFile));
                VCInstallDir = VSInstallDir ~ r"VC\";
                if (!ver.empty)
                    VCToolsInstallDir = VCInstallDir ~ r"Tools\MSVC\" ~ ver ~ r"\";
```

```
            }
            catch(Exception)
            {
            }
        }
        if (VCInstallDir.empty)
        {
            if(char* pe = getenv("VCINSTALLDIR"))
                VCInstallDir = fromMBSz(cast(immutable)pe);
            else
            {
                scope RegKey keyVS = new RegKey(hConfigKey, regConfigRoot ~ "\\Setup\\VC",
false);
                VCInstallDir = toUTF8(keyVS.GetString("ProductDir"));
            }
            VCInstallDir = normalizeDir(VCInstallDir);
        }
    }

    string getVCDir(string sub, bool x64, bool expand = false)
    {
        string dir;
        if (!VCToolsInstallDir.empty)
        {
            dir = (expand ? VCToolsInstallDir : "$(VCTOOLSINSTALLDIR)");
            if (sub.startsWith("bin"))
                sub = (x64 ? "bin\\HostX86\\x64" : "bin\\HostX86\\x86") ~ sub[3 .. $];
            if (sub.startsWith("lib"))
                sub = (x64 ? "lib\\x64" : "lib\\x86") ~ sub[3 .. $];
        }
        else
        {
            dir = (expand ? VCInstallDir : "$(VCINSTALLDIR)");
            if (sub.startsWith("lib") && x64)
                sub = "lib\\amd64" ~ sub[3 .. $];
        }
        return dir ~ sub;
    }
```

```
    void detectDMDInstallDir()
    {
        scope RegKey keyVD = new RegKey(HKEY_LOCAL_MACHINE,
"SOFTWARE\\VisualD", false);
        DMD.InstallDir = toUTF8(keyVD.GetString("DMDInstallDir"));
        if(DMD.InstallDir.empty)
        {
            scope RegKey keyDMD = new RegKey(HKEY_LOCAL_MACHINE,
"SOFTWARE\\DMD", false);
            string dir = toUTF8(keyDMD.GetString("InstallationFolder"));
            if(!dir.empty)
                DMD.InstallDir = dir ~ "\\dmd2";
        }
    }

    bool initFromRegistry()
    {
        if(!getRegistryRoot())
            return false;

        wstring dllPath = GetDLLName(g_hInst);
        VisualDInstallDir = normalizeDir(dirName(toUTF8(dllPath)));

        wstring idePath = GetDLLName(null);
        DevEnvDir = normalizeDir(dirName(toUTF8(idePath)));

        bool rc = true;
        try
        {
            wstring defUserTypesSpec = "Object string wstring dstring ClassInfo\n" ~
                                       "hash_t ptrdiff_t size_t sizediff_t";
            // get defaults from global config
            scope RegKey keyToolOpts = new RegKey(hConfigKey, regConfigRoot ~
regPathToolsOptions, false);
            scope RegKey keyUserOpts = new RegKey(hUserKey, regUserRoot ~
regPathToolsOptions, false);

            detectWindowsSDKDir();
            detectUCRT();
```

```
        detectVSInstallDir();
        detectVCInstallDir();
        detectDMDInstallDir();

        //UtilMessageBox("getVCDir = " ~ getVCDir("lib\\legacy_stdio_definitions.lib", true,
true)
        //UtilMessageBox("VSInstallDir = "~VSInstallDir ~"\n" ~
        //               "VCInstallDir = "~VCInstallDir ~"\n" ~
        //               "VCToolsInstallDir = "~VCToolsInstallDir ~ "\n" ~
        //               "regConfig = " ~ to!string(regConfigRoot) ~ "\n" ~
        //               "regUser = " ~ to!string(regUserRoot)
        //               , MB_OK, "Visual D - init");

        wstring getWStringOpt(wstring tag, wstring def = null)
        {
            wstring ws = keyToolOpts.GetString(tag, def);
            return keyUserOpts.GetString(tag, ws);
        }
        string getStringOpt(wstring tag, wstring def = null)
        {
            return toUTF8(getWStringOpt(tag, def));
        }
        string getPathsOpt(wstring tag, string def = null)
        {
            return replaceSemiCrLf(toUTF8(getWStringOpt(tag, to!wstring(def))));
        }
        int getIntOpt(wstring tag, int def = 0)
        {
            int v = keyToolOpts.GetDWORD(tag, def);
            return keyUserOpts.GetDWORD(tag, v);
        }
        bool getBoolOpt(wstring tag, bool def = false)
        {
            return getIntOpt(tag, def ? 1 : 0) != 0;
        }

        ColorizeVersions    = getBoolOpt("ColorizeVersions", true);
        ColorizeCoverage    = getBoolOpt("ColorizeCoverage", true);
        showCoverageMargin  = getBoolOpt("showCoverageMargin", false);
```

```
        timeBuilds          = getBoolOpt("timeBuilds", false);
        elasticSpace        = getBoolOpt("elasticSpace", true);
        sortProjects        = getBoolOpt("sortProjects", true);
        stopSolutionBuild   = getBoolOpt("stopSolutionBuild", false);
        showUptodateFailure = getBoolOpt("showUptodateFailure", false);
        demangleError       = getBoolOpt("demangleError", true);
        optlinkDeps         = getBoolOpt("optlinkDeps", true);
        excludeFileDeps     = getPathsOpt("excludeFileDeps");
        static if(enableShowMemUsage)
            showMemUsage    = getBoolOpt("showMemUsage", false);
        autoOutlining       = getBoolOpt("autoOutlining", true);
        deleteFiles         = cast(byte) getIntOpt("deleteFiles", 0);
        parseSource         = getBoolOpt("parseSource", true);
        expandFromSemantics = getBoolOpt("expandFromSemantics", true);
        expandFromBuffer    = getBoolOpt("expandFromBuffer", true);
        expandFromJSON      = getBoolOpt("expandFromJSON", true);
        expandTrigger       = cast(byte) getIntOpt("expandTrigger", 0);
        showTypeInTooltip   = getBoolOpt("showTypeInTooltip2", true); // changed default
        semanticGotoDef     = getBoolOpt("semanticGotoDef", true);
        pasteIndent         = getBoolOpt("pasteIndent", true);

        scope RegKey keyDParser = new RegKey(HKEY_CLASSES_ROOT, "CLSID\\
{002a2de9-8bb6-484d-AA05-7e4ad4084715}", false);
        useDParser          = true; // getBoolOpt("useDParser2", keyDParser.key !is null);
        mixinAnalysis       = getBoolOpt("mixinAnalysis", false);
        UFCSExpansions      = getBoolOpt("UFCSExpansions", true);
        sortExpMode         = getBoolOpt("sortExpMode", 0);
        exactExpMatch       = getBoolOpt("exactExpMatch", true);

        string getDefaultLibPathCOFF64()
        {
            string libpath = getVCDir ("lib", true);
            string dir = replaceGlobalMacros(libpath);
            if(std.file.exists(dir ~ "\\legacy_stdio_definitions.lib"))
                libpath ~= "\n$(UCRTSdkDir)Lib\\$(UCRTVersion)\\ucrt\\x64";

            if(WindowsSdkDir.length)
            {
                if(std.file.exists(WindowsSdkDir ~ r"lib\x64\kernel32.lib"))
```

```
            libpath ~= "\n$(WindowsSdkDir)lib\\x64";
        else if(std.file.exists(WindowsSdkDir ~ r"Lib\win8\um\x64\kernel32.lib")) // SDK
8.0
            libpath ~= "\n$(WindowsSdkDir)Lib\\win8\\um\\x64";
        else if(std.file.exists(WindowsSdkDir ~ r"Lib\winv6.3\um\x64\kernel32.lib")) // SDK
8.1
            libpath ~= "\n$(WindowsSdkDir)Lib\\winv6.3\\um\\x64";
    }
    return libpath;
}

string getDefaultLibPathCOFF32()
{
    string libpath = getVCDir ("lib", false);
    string dir = replaceGlobalMacros(libpath);
    if(std.file.exists(dir ~ "\\legacy_stdio_definitions.lib"))
        libpath ~= "\n$(UCRTSdkDir)Lib\\$(UCRTVersion)\\ucrt\\x86";

    if(WindowsSdkDir.length)
    {
        if(std.file.exists(WindowsSdkDir ~ r"lib\kernel32.lib"))
            libpath ~= "\n$(WindowsSdkDir)lib";
        else if(std.file.exists(WindowsSdkDir ~ r"Lib\win8\um\x86\kernel32.lib")) // SDK
8.0
            libpath ~= "\n$(WindowsSdkDir)Lib\\win8\\um\\x86";
        else if(std.file.exists(WindowsSdkDir ~ r"Lib\winv6.3\um\x86\kernel32.lib")) // SDK
8.1
            libpath ~= "\n$(WindowsSdkDir)Lib\\winv6.3\\um\\x86";
    }
    return libpath;
}

// overwrite by user config
void readCompilerOptions(string compiler)(ref CompilerDirectories opt)
{
    enum bool dmd = compiler == "DMD";
    enum string prefix = dmd ? "" : compiler ~ ".";
    if (auto dir = getStringOpt(compiler ~ "InstallDir"))
        opt.InstallDir = dir;
```

```
            opt.ExeSearchPath   = getPathsOpt(prefix ~ "ExeSearchPath", opt.ExeSearchPath);
            opt.LibSearchPath   = getPathsOpt(prefix ~ "LibSearchPath", opt.LibSearchPath);
            opt.ImpSearchPath   = getPathsOpt(prefix ~ "ImpSearchPath", opt.ImpSearchPath);
            opt.DisasmCommand   = getPathsOpt(prefix ~ "DisasmCommand",
opt.DisasmCommand);
            opt.ExeSearchPath64 = getPathsOpt(prefix ~ "ExeSearchPath64",
opt.ExeSearchPath64);
            opt.LibSearchPath64 = getPathsOpt(prefix ~ "LibSearchPath64",
opt.LibSearchPath64);
            opt.DisasmCommand64 = getPathsOpt(prefix ~ "DisasmCommand64",
opt.DisasmCommand64);

            wstring linkPath = to!wstring(getVCDir("bin\\link.exe", false));
            opt.overrideIni64     = getBoolOpt(prefix ~ "overrideIni64", dmd);
            opt.overrideLinker64  = getStringOpt(prefix ~ "overrideLinker64", dmd ? linkPath :
"");
            opt.overrideOptions64 = getStringOpt(prefix ~ "overrideOptions64");

            if (dmd)
            {
                opt.ExeSearchPath32coff   = getPathsOpt(prefix ~ "ExeSearchPath32coff",
opt.ExeSearchPath32coff);
                opt.LibSearchPath32coff   = getPathsOpt(prefix ~ "LibSearchPath32coff",
opt.LibSearchPath32coff);
                opt.DisasmCommand32coff   = getPathsOpt(prefix ~ "DisasmCommand32coff",
opt.DisasmCommand32coff);
                opt.overrideIni32coff     = getBoolOpt(prefix ~ "overrideIni32coff", true);
                opt.overrideLinker32coff  = getStringOpt(prefix ~ "overrideLinker32coff", linkPath);
                opt.overrideOptions32coff = getStringOpt(prefix ~ "overrideOptions32coff");
            }
        }
        // put dmd bin folder at the end to avoid trouble with link.exe (dmd does not need
search path)
        // $(WindowsSdkDir)\bin needed for rc.exe
        // $(VCInstallDir)\bin needed to compile C + link.exe + DLLs
        // $(VSINSTALLDIR)\Common7\IDE needed for some VS versions for cv2pdb
        string sdkBinDir = "$(WindowsSdkDir)bin";
        if (std.file.exists(WindowsSdkDir ~ "bin\\x86\\rc.exe")) // path changed with Windows 8
```

```
SDK
            sdkBinDir ~= "\\x86";
        DMD.ExeSearchPath       = getVCDir("bin", false) ~
r";$(VSINSTALLDIR)Common7\IDE;" ~ sdkBinDir ~ ";$(DMDInstallDir)windows\\bin";
        DMD.ExeSearchPath64     = DMD.ExeSearchPath;
        DMD.ExeSearchPath32coff = DMD.ExeSearchPath;
        GDC.ExeSearchPath       = r"$(GDCInstallDir)bin;$(VSINSTALLDIR)Common7\IDE;" ~
sdkBinDir;
        GDC.ExeSearchPath64     = GDC.ExeSearchPath;
        LDC.ExeSearchPath       = r"$(LDCInstallDir)bin;" ~ getVCDir("bin", false) ~
r";$(VSINSTALLDIR)Common7\IDE;" ~ sdkBinDir;
        LDC.ExeSearchPath64     = r"$(LDCInstallDir)bin;" ~ getVCDir("bin", true)  ~
r";$(VSINSTALLDIR)Common7\IDE;" ~ sdkBinDir;

        DMD.LibSearchPath64     = getDefaultLibPathCOFF64();
        LDC.LibSearchPath64     = DMD.LibSearchPath64;
        DMD.LibSearchPath32coff = getDefaultLibPathCOFF32();
        LDC.LibSearchPath       = DMD.LibSearchPath32coff;

        DMD.DisasmCommand       = `"obj2asm" -x "$(InputPath)" >"$(TargetPath)"`;
        DMD.DisasmCommand64     = `"` ~ getVCDir("bin\\dumpbin", true)  ~ `"
/disasm:nobytes "$(InputPath)" >"$(TargetPath)"`;
        DMD.DisasmCommand32coff = `"` ~ getVCDir("bin\\dumpbin", false) ~ `"
/disasm:nobytes "$(InputPath)" >"$(TargetPath)"`;

        GDC.DisasmCommand   = DMD.DisasmCommand32coff;
        LDC.DisasmCommand   = DMD.DisasmCommand32coff;
        GDC.DisasmCommand64 = DMD.DisasmCommand64;
        LDC.DisasmCommand64 = DMD.DisasmCommand64;

        readCompilerOptions!"DMD"(DMD);
        readCompilerOptions!"GDC"(GDC);
        readCompilerOptions!"LDC"(LDC);

        JSNSearchPath     = getPathsOpt("JSNSearchPath");
        IncSearchPath     = getStringOpt("IncSearchPath", r"$(WindowsSdkDir)include;"w ~
to!wstring(getVCDir("include", false)));
        VDServerIID       = getStringOpt("VDServerIID");
        compileAndRunOpts = getStringOpt("compileAndRunOpts", "-unittest");
```

```
        compileAndDbgOpts = getStringOpt("compileAndDbgOpts", "-g");
        compileAndDbgEngine = getIntOpt("compileAndDbgEngine", 0);

        string execDirs   = getStringOpt("coverageExecutionDirs", "");
        coverageExecutionDirs = split(execDirs, ";");
        string buildDirs  = getStringOpt("coverageBuildDirs", "");
        coverageBuildDirs = split(buildDirs, ";");

        UserTypesSpec     = getStringOpt("UserTypesSpec", defUserTypesSpec);
        UserTypes = parseUserTypes(UserTypesSpec);

        lastColorizeCoverage = ColorizeCoverage;
        lastColorizeVersions = ColorizeVersions;
        lastUseDParser       = useDParser;

        updateDefaultColors();

        if(VDServerIID.length > 0)
            gServerClassFactory_iid = uuid(VDServerIID);
        else
            updateVDServer();

        CHierNode.setContainerIsSorted(sortProjects);
    }
    catch(Exception e)
    {
        writeToBuildOutputPane(e.msg);
        rc = false;
    }

    return rc;
}

void updateVDServer()
{
    if(useDParser)
        gServerClassFactory_iid = DParserClassFactory_iid;
    else
        gServerClassFactory_iid = VDServerClassFactory_iid;
```

```
    }

    bool saveToRegistry()
    {
        if(!getRegistryRoot())
            return false;

        try
        {
            scope RegKey keyToolOpts = new RegKey(hUserKey, regUserRoot ~
regPathToolsOptions);
            keyToolOpts.Set("DMDInstallDir",     toUTF16(DMD.InstallDir));
            keyToolOpts.Set("GDCInstallDir",     toUTF16(GDC.InstallDir));
            keyToolOpts.Set("LDCInstallDir",     toUTF16(LDC.InstallDir));
            keyToolOpts.Set("ExeSearchPath",     toUTF16(DMD.ExeSearchPath));
            keyToolOpts.Set("LibSearchPath",     toUTF16(DMD.LibSearchPath));
            keyToolOpts.Set("ImpSearchPath",     toUTF16(DMD.ImpSearchPath));
            keyToolOpts.Set("DisasmCommand",     toUTF16(DMD.DisasmCommand));
            keyToolOpts.Set("GDC.ExeSearchPath", toUTF16(GDC.ExeSearchPath));
            keyToolOpts.Set("GDC.LibSearchPath", toUTF16(GDC.LibSearchPath));
            keyToolOpts.Set("GDC.ImpSearchPath", toUTF16(GDC.ImpSearchPath));
            keyToolOpts.Set("GDC.DisasmCommand", toUTF16(GDC.DisasmCommand));
            keyToolOpts.Set("LDC.ExeSearchPath", toUTF16(LDC.ExeSearchPath));
            keyToolOpts.Set("LDC.LibSearchPath", toUTF16(LDC.LibSearchPath));
            keyToolOpts.Set("LDC.ImpSearchPath", toUTF16(LDC.ImpSearchPath));
            keyToolOpts.Set("LDC.DisasmCommand", toUTF16(LDC.DisasmCommand));
            keyToolOpts.Set("JSNSearchPath",     toUTF16(JSNSearchPath));
            keyToolOpts.Set("IncSearchPath",     toUTF16(IncSearchPath));
            keyToolOpts.Set("UserTypesSpec",     toUTF16(UserTypesSpec));

            keyToolOpts.Set("ExeSearchPath64",     toUTF16(DMD.ExeSearchPath64));
            keyToolOpts.Set("LibSearchPath64",     toUTF16(DMD.LibSearchPath64));
            keyToolOpts.Set("DisasmCommand64",     toUTF16(DMD.DisasmCommand64));
            keyToolOpts.Set("overrideIni64",       DMD.overrideIni64);
            keyToolOpts.Set("overrideLinker64",    toUTF16(DMD.overrideLinker64));
            keyToolOpts.Set("overrideOptions64",   toUTF16(DMD.overrideOptions64));

            keyToolOpts.Set("ExeSearchPath32coff",     toUTF16(DMD.ExeSearchPath32coff));
            keyToolOpts.Set("LibSearchPath32coff",     toUTF16(DMD.LibSearchPath32coff));
```

```
        keyToolOpts.Set("DisasmCommand32coff",
toUTF16(DMD.DisasmCommand32coff));
        keyToolOpts.Set("overrideIni32coff",       DMD.overrideIni32coff);
        keyToolOpts.Set("overrideLinker32coff",    toUTF16(DMD.overrideLinker32coff));
        keyToolOpts.Set("overrideOptions32coff",   toUTF16(DMD.overrideOptions32coff));

        keyToolOpts.Set("GDC.ExeSearchPath64", toUTF16(GDC.ExeSearchPath64));
        keyToolOpts.Set("GDC.LibSearchPath64", toUTF16(GDC.LibSearchPath64));
        keyToolOpts.Set("GDC.DisasmCommand64", toUTF16(GDC.DisasmCommand64));
        keyToolOpts.Set("LDC.ExeSearchPath64", toUTF16(LDC.ExeSearchPath64));
        keyToolOpts.Set("LDC.LibSearchPath64", toUTF16(LDC.LibSearchPath64));
        keyToolOpts.Set("LDC.DisasmCommand64", toUTF16(LDC.DisasmCommand64));

        keyToolOpts.Set("ColorizeVersions",    ColorizeVersions);
        keyToolOpts.Set("ColorizeCoverage",    ColorizeCoverage);
        keyToolOpts.Set("showCoverageMargin",  showCoverageMargin);
        keyToolOpts.Set("excludeFileDeps",     toUTF16(excludeFileDeps));
        keyToolOpts.Set("timeBuilds",          timeBuilds);
        keyToolOpts.Set("elasticSpace",        elasticSpace);
        keyToolOpts.Set("sortProjects",        sortProjects);
        keyToolOpts.Set("stopSolutionBuild",   stopSolutionBuild);
        keyToolOpts.Set("showUptodateFailure", showUptodateFailure);
        keyToolOpts.Set("demangleError",       demangleError);
        keyToolOpts.Set("optlinkDeps",         optlinkDeps);
        keyToolOpts.Set("showMemUsage",        showMemUsage);
        keyToolOpts.Set("autoOutlining",       autoOutlining);
        keyToolOpts.Set("deleteFiles",         deleteFiles);
        keyToolOpts.Set("parseSource",         parseSource);
        keyToolOpts.Set("expandFromSemantics", expandFromSemantics);
        keyToolOpts.Set("expandFromBuffer",    expandFromBuffer);
        keyToolOpts.Set("expandFromJSON",      expandFromJSON);
        keyToolOpts.Set("expandTrigger",       expandTrigger);
        keyToolOpts.Set("showTypeInTooltip2",  showTypeInTooltip);
        keyToolOpts.Set("semanticGotoDef",     semanticGotoDef);
        keyToolOpts.Set("useDParser2",         useDParser);
        keyToolOpts.Set("mixinAnalysis",       mixinAnalysis);
        keyToolOpts.Set("UFCSExpansions",      UFCSExpansions);
        keyToolOpts.Set("sortExpMode",         sortExpMode);
        keyToolOpts.Set("exactExpMatch",       exactExpMatch);
```

```
    keyToolOpts.Set("pasteIndent",         pasteIndent);
    keyToolOpts.Set("compileAndRunOpts",   toUTF16(compileAndRunOpts));
    keyToolOpts.Set("compileAndDbgOpts",   toUTF16(compileAndDbgOpts));
    keyToolOpts.Set("compileAndDbgEngine", compileAndDbgEngine);

    keyToolOpts.Set("coverageExecutionDirs", toUTF16(join(coverageExecutionDirs, ";")));
    keyToolOpts.Set("coverageBuildDirs",   toUTF16(join(coverageBuildDirs, ";")));

    CHierNode.setContainerIsSorted(sortProjects);
}
catch(Exception e)
{
    writeToBuildOutputPane(e.msg);
    return false;
}

bool updateColorizer = false;
int[wstring] types = parseUserTypes(UserTypesSpec);
if(types != UserTypes)
{
    UserTypes = types;
    updateColorizer = true;
}
if(lastColorizeVersions != ColorizeVersions)
{
    lastColorizeVersions = ColorizeVersions;
    updateColorizer = true;
}
if(lastColorizeCoverage != ColorizeCoverage)
{
    lastColorizeCoverage = ColorizeCoverage;
    updateColorizer = true;
}
if(updateColorizer)
    if(auto svc = Package.s_instance.mLangsvc)
        svc.UpdateColorizer(true);

if(lastUseDParser != useDParser)
{
```

```
        updateVDServer();
        lastUseDParser = useDParser;
        VDServerClient.restartServer = true;
    }
    else if(!expandFromSemantics)
        Package.GetLanguageService().ClearSemanticProject();

    Package.scheduleUpdateLibrary();
    return true;
}

void addReplacements(ref string[string] replacements)
{
    replacements["DMDINSTALLDIR"] = normalizeDir(DMD.InstallDir);
    replacements["GDCINSTALLDIR"] = normalizeDir(GDC.InstallDir);
    replacements["LDCINSTALLDIR"] = normalizeDir(LDC.InstallDir);
    replacements["WINDOWSSDKDIR"] = WindowsSdkDir;
    replacements["UCRTSDKDIR"] = UCRTSdkDir;
    replacements["UCRTVERSION"] = UCRTVersion;
    replacements["DEVENVDIR"] = DevEnvDir;
    replacements["VCINSTALLDIR"] = VCInstallDir;
    replacements["VCTOOLSINSTALLDIR"] = VCToolsInstallDir;
    replacements["VSINSTALLDIR"] = VSInstallDir;
    replacements["VISUALDINSTALLDIR"] = VisualDInstallDir;
}

string replaceGlobalMacros(string s)
{
    if(s.indexOf('$') < 0)
        return s;

    string[string] replacements;
    addReplacements(replacements);
    return replaceMacros(s, replacements);
}

string findInPath(string exe)
{
    string searchpaths = replaceGlobalMacros(DMD.ExeSearchPath);
```

```
    string[] paths = tokenizeArgs(searchpaths, true, false);
    if(char* p = getenv("PATH"))
        paths ~= tokenizeArgs(to!string(p), true, false);

    foreach(path; paths)
    {
        path = unquoteArgument(path);
        path = normalizeDir(path);
        if(std.file.exists(path ~ exe))
            return path;
    }
    return null;
}

string findDmdBinDir(string dmdpath = null)
{
    if(dmdpath.length && std.file.exists(dmdpath))
        return normalizeDir(dirName(dmdpath));

    string installdir = normalizeDir(DMD.InstallDir);
    string bindir = installdir ~ "windows\\bin\\";
    if(std.file.exists(bindir ~ "dmd.exe"))
        return bindir;

    string dmd = findInPath("dmd.exe");
    return empty(dmd) ? null : dirName(dmd);
}

string findScIni(string workdir, string dmdpath, bool optlink)
{
    string inifile;
    if(workdir.length)
        inifile = buildPath(workdir, "sc.ini");

    if(inifile.empty || !std.file.exists(inifile))
    {
        inifile = null;
        if(auto home = getenv("HOME"))
            inifile = buildPath(fromMBSz(cast(immutable)home), "sc.ini");
```

```
        }
        if(inifile.empty || !std.file.exists(inifile))
        {
            inifile = null;
            string dmddir = findDmdBinDir(dmdpath);
            if(!dmddir.empty)
            {
                if(optlink)
                    dmddir = dmddir; // TODO: in case link is elsewhere it uses a different sc.ini
                inifile = buildPath(dmddir, "sc.ini");
            }
        }
        if(inifile.empty || !std.file.exists(inifile))
            inifile = null;
        return inifile;
    }

    string getLinkerPath(bool x64, bool mscoff, string workdir, string dmdpath, string *libs = null,
string* options = null)
    {
        string path = "link.exe";
        string inifile = findScIni(workdir, dmdpath, false);
        if(!inifile.empty)
        {
            string[string] env = [ "@P" : dirName(inifile) ];
            addReplacements(env);
            string[string][string] ini = parseIni(inifile);

            if(auto pEnv = "Environment" in ini)
                env = expandIniSectionEnvironment((*pEnv)[""], env);

            string envArch = x64 ? "Environment64" : mscoff ? "Environment32mscoff" :
"Environment32";
            if(auto pEnv = envArch in ini)
                env = expandIniSectionEnvironment((*pEnv)[""], env);

            if(string* pLink = "LINKCMD" in env)
                path = *pLink;
            if(x64)
```

```
        {
            if(DMD.overrideIni64)
                path = DMD.overrideLinker64;
            else if(string* pLink = "LINKCMD64" in env)
                path = *pLink;
        }
        else if(mscoff)
        {
            if(DMD.overrideIni32coff)
                path = DMD.overrideLinker32coff;
        }

        if(options)
        {
            if(x64 && DMD.overrideIni64)
                *options = DMD.overrideOptions64;
            else if(!x64 && mscoff && DMD.overrideIni32coff)
                *options = DMD.overrideOptions32coff;
            else if(string* pFlags = "DFLAGS" in env)
                *options = *pFlags;
        }
        if(libs)
            if(string* pLibs = "LIB" in env)
                *libs = *pLibs;
    }
    return path;
}

static string[] getOptionImportPaths(string opts, string workdir)
{
    string[] imports;
    string[] args = tokenizeArgs(opts);
    args = expandResponseFiles(args, workdir);
    foreach(arg; args)
    {
        arg = unquoteArgument(arg);
        if(arg.startsWith("-I"))
            imports ~= removeDotDotPath(normalizeDir(arg[2..$]));
    }
```

```
    return imports;
}

string[] getIniImportPaths()
{
    string[] imports;
    string bindir = findDmdBinDir();
    string inifile = bindir ~ "sc.ini";
    if(std.file.exists(inifile))
    {
        string[string][string] ini = parseIni(inifile);
        if(auto pEnv = "Environment" in ini)
            if(string* pFlags = "DFLAGS" in *pEnv)
            {
                string opts = replace(*pFlags, "%@P%", bindir);
                imports ~= getOptionImportPaths(opts, bindir);
            }
    }
    return imports;
}

string[] getImportPaths()
{
    string[] imports = getIniImportPaths();
    string searchpaths = replaceGlobalMacros(DMD.ImpSearchPath);
    string[] args = tokenizeArgs(searchpaths);
    foreach(arg; args)
        imports ~= removeDotDotPath(normalizeDir(unquoteArgument(arg)));

    return imports;
}

string[] getJSONPaths()
{
    string[] jsonpaths;
    string searchpaths = replaceGlobalMacros(JSNSearchPath);
    string[] args = tokenizeArgs(searchpaths);
    foreach(arg; args)
        jsonpaths ~= normalizeDir(unquoteArgument(arg));
```

```
    return jsonpaths;
}

string[] getJSONFiles()
{
    string[] jsonpaths = getJSONPaths();

    string[] jsonfiles;
    foreach(path; jsonpaths)
    {
        if(isExistingDir(path))
            foreach (string name; dirEntries(path, SpanMode.shallow))
                if (globMatch(baseName(name), "*.json"))
                    addunique(jsonfiles, name);
    }
    return jsonfiles;
}

string[] getDepsExcludePaths()
{
    string[] exclpaths;
    string paths = replaceGlobalMacros(excludeFileDeps);
    string[] args = tokenizeArgs(paths);
    foreach(arg; args)
        if (!arg.empty)
            exclpaths ~= normalizePath(unquoteArgument(arg));
    return exclpaths;
}

void logSettingsTree(IVsProfileSettingsTree settingsTree)
{
    logIndent(1); scope(exit) logIndent(-1);
    BSTR bname;
    string name, desc, cat, regname, nameForId, fullPath, pkg;
    if(SUCCEEDED(settingsTree.GetDisplayName(&bname)))
        name = detachBSTR(bname);
    if(SUCCEEDED(settingsTree.GetDescription(&bname)))
        desc = detachBSTR(bname);
    if(SUCCEEDED(settingsTree.GetCategory(&bname)))
```

```
        cat = detachBSTR(bname);
    if(SUCCEEDED(settingsTree.GetRegisteredName(&bname)))
        regname = detachBSTR(bname);
    if(SUCCEEDED(settingsTree.GetNameForID(&bname)))
        nameForId = detachBSTR(bname);
    if(SUCCEEDED(settingsTree.GetFullPath(&bname)))
        fullPath = detachBSTR(bname);
    if(SUCCEEDED(settingsTree.GetPackage(&bname)))
        pkg = detachBSTR(bname);
    logCall("Name: " ~ name ~ ", Desc: " ~ desc ~ ", Cat: " ~ cat ~ ", regname: " ~ regname,
", nameForId: " ~ nameForId ~ ", fullpath: " ~ fullPath ~ ", pkg: " ~ pkg);

    int count;
    if(SUCCEEDED(settingsTree.GetChildCount(&count)))
        for(int i = 0; i < count; i++)
        {
            IVsProfileSettingsTree child;
            if(SUCCEEDED(settingsTree.GetChild(i, &child)))
            {
                scope(exit) release(child);
                logSettingsTree(child);
            }
        }
}

version(none)
string[] getVCLibraryPaths()
{
    IVsProfileDataManager pdm = queryService!
(SVsProfileDataManager,IVsProfileDataManager)();
    if(!pdm)
        return null;
    scope(exit) release(pdm);

    IVsProfileSettingsTree settingsTree;
    HRESULT hr = pdm.GetSettingsForExport(&settingsTree);
    if(SUCCEEDED(hr))
    {
        scope(exit) release(settingsTree);
```

```
        logSettingsTree(settingsTree);
    }
    //pdm.ShowProfilesUI();
    return null;
}

bool buildPhobosBrowseInfo()
{
    IVsOutputWindowPane pane = getVisualDOutputPane();
    if(!pane)
        return false;
    scope(exit) release(pane);

    string[] jsonPaths = getJSONPaths();
    string jsonPath;
    if(jsonPaths.length)
        jsonPath = jsonPaths[0];
    if(jsonPath.length == 0)
    {
        JSNSearchPath ~= "\"$(APPDATA)\\VisualD\\json\\\"";
        saveToRegistry();
        jsonPath = getJSONPaths()[0];
    }

    pane.Clear();
    pane.Activate();
    string msg = "Building phobos JSON browse information files to " ~ jsonPath ~ "\n";
    pane.OutputString(toUTF16z(msg));

    if(!std.file.exists(jsonPath))
    {
        try
        {
            mkdirRecurse(jsonPath[0..$-1]); // normalized dir has trailing slash
        }
        catch(Exception)
        {
            return OutputErrorString(msg = "cannot create directory " ~ jsonPath);
        }
```

```
    }

    string[] imports = getIniImportPaths();
    foreach(s; imports)
        pane.OutputString(toUTF16z("Using import " ~ s ~ "\n"));

    string cmdfile = jsonPath ~ "buildjson.bat";
    string dmddir = findDmdBinDir();
    string dmdpath = dmddir ~ "dmd.exe";
    if(!std.file.exists(dmdpath))
        return OutputErrorString(msg = "dmd.exe not found in DMDInstallDir=" ~
DMD.InstallDir ~ " or through PATH");

    foreach(s; imports)
    {
        string[] files;
        string cmdline = "@echo off\n";
        string jsonfile;
        string opts = " -d -c -o-";

        if(std.file.exists(s ~ "std\\algorithm.d") || std.file.exists(s ~ "std\\algorithm\\package.d"))
// D2
        {
            files ~= findDRuntimeFiles(s, "std", true);
            files ~= findDRuntimeFiles(s, "etc\\c", true);
            jsonfile = jsonPath ~ "phobos.json";
        }
        if(std.file.exists(s ~ "std\\gc.d")) // D1
        {
            files ~= findDRuntimeFiles(s, "std", false);
            files ~= findDRuntimeFiles(s, "std\\c", false);
            files ~= findDRuntimeFiles(s, "std\\c\\windows", false);
            files ~= findDRuntimeFiles(s, "std\\windows", false);
            jsonfile = jsonPath ~ "phobos1.json";
        }
        if(std.file.exists(s ~ "object.di") || std.file.exists(s ~ "object.d"))
        {
            opts ~= " -I" ~ buildPath(s, "..\\src"); // needed since dmd 2.059
            if (std.file.exists(s ~ "object.di"))
```

```
                files ~= "object.di";
            else
                files ~= "object.d"; // dmd >=2.068 no longer has object.di
            files ~= findDRuntimeFiles(s, "core", true);
            files ~= findDRuntimeFiles(s, "std", false);  // D1?
            jsonfile = jsonPath ~ "druntime.json";
        }

        if(files.length)
        {
            string sfiles = std.string.join(files, " ");
            cmdline ~= quoteFilename(dmdpath) ~ opts ~ " -Xf" ~ quoteFilename(jsonfile) ~ " " ~
sfiles ~ "\n\n";
            pane.OutputString(toUTF16z("Building " ~ jsonfile ~ " from import " ~ s ~ "\n"));
            if(!launchBuildPhobos(s, cmdfile, cmdline, pane))
                pane.OutputString(toUTF16z("Building " ~ jsonfile ~ " failed!\n"));
            else
                pane.OutputString(toUTF16z("Building " ~ jsonfile ~ " successful!\n"));
        }
    }
    return true;
}

string findCoverageFile(string srcfile)
{
    import stdext.path;
    import std.path;
    import std.file;
    import std.string;

    string lstname = std.path.stripExtension(srcfile) ~ ".lst";
    if(std.file.exists(lstname) && std.file.isFile(lstname))
        return lstname;

    string srcpath = stripExtension(toLower(makeFilenameCanonical(srcfile, "")));

    foreach(dir; coverageExecutionDirs)
    {
        if(!std.file.exists(dir) || !std.file.isDir(dir))
```

```
    continue;

foreach(string f; dirEntries(dir, SpanMode.shallow))
{
    char[] fn = baseName(f).dup;
    toLowerInPlace(fn);
    auto ext = extension(fn);
    if(ext != ".lst")
        continue;

    // assume no '-' in file name, cov replaced '\\' with these
    bool isAbs = false;
    if(std.ascii.isAlpha(fn[0]) && fn[1] == '-' && fn[2] == '-')
    {
        // absolute path
        fn[1] = ':';
        isAbs = true;
    }
    for(size_t i = 0; i < fn.length; i++)
        if(fn[i] == '-')
            fn[i] = '\\';

    string fs = to!string(fn);
    if(isAbs)
    {
        fs = removeDotDotPath(fs);
        if(fs[0 .. $-4] == srcpath)
            return std.path.buildPath(dir, f);
    }
    else
    {
        foreach(bdir; coverageBuildDirs)
        {
            string bfile = toLower(makeFilenameCanonical(fs, bdir));
            if(bfile[0 .. $-4] == srcpath)
                return std.path.buildPath(dir, f);
        }
    }
}
```

```
    }
    return null;
}

void DeleteCoverageFiles()
{
    string[] dirs = coverageExecutionDirs;
    Source[] srcs = Package.GetLanguageService().GetSources();
    foreach(src; srcs)
        dirs.addunique(dirName(src.GetFileName()));

    foreach(dir; dirs)
        if(std.file.exists(dir) && std.file.isDir(dir))
        {
            string[] lstfiles;
            foreach(f; std.file.dirEntries(dir, SpanMode.shallow))
                if(icmp(extension(f.name), ".lst") == 0)
                    lstfiles ~= f;
            foreach(lst; lstfiles)
                collectException(std.file.remove(lst));
        }

    coverageExecutionDirs = null;
}

void addExecutionPath(string dir, string workdir = null)
{
    dir = makeDirnameCanonical(dir, workdir);
    adduniqueLRU(coverageExecutionDirs, dir, 4);
}

void addBuildPath(string dir, string workdir = null)
{
    dir = makeDirnameCanonical(dir, workdir);
    adduniqueLRU(coverageBuildDirs, dir, 4);
}

////////////////////////////////////////////////////////////////////////////
void updateDefaultColors()
```

```
    {
        scope RegKey keyUserOpts = new RegKey(hUserKey, regConfigRoot ~
regPathToolsOptions, false);
        bool wasDark = keyUserOpts.GetDWORD("lastThemeWasDark") != 0;
        bool isDark = isDarkTheme();
        if (wasDark != isDark)
        {
            scope RegKey keyUserOptsWr = new RegKey(hUserKey, regConfigRoot ~
regPathToolsOptions, true);
            removeColorCache();
            keyUserOptsWr.Set("lastThemeWasDark", isDark);
        }
        LanguageService.updateThemeColors();
    }


    bool isDarkTheme()
    {
        scope RegKey keyUserOpts = new RegKey(hUserKey, regUserRoot ~ r"\General", false);
        string theme = toUTF8(keyUserOpts.GetString("CurrentTheme")).toLower;
        if (theme == "1ded0138-47ce-435e-84ef-9ec1f439b749" || theme == "{1ded0138-47ce-
435e-84ef-9ec1f439b749}")
            return true;

        // VS2015
        scope RegKey keyUserOpts15 = new RegKey(hUserKey, regUserRoot ~
r"\ApplicationPrivateSettings\Microsoft\VisualStudio", false);
        string theme15 = toUTF8(keyUserOpts15.GetString("ColorTheme")).toLower;
        return theme15.endsWith("1ded0138-47ce-435e-84ef-9ec1f439b749");
    }

    bool removeColorCache()
    {
        auto hr = RegDeleteRecursive(hUserKey, regUserRoot ~ r"\FontAndColors\Cache\
{E0187991-B458-4F7E-8CA9-42C9A573B56C}");
        return SUCCEEDED(hr);
    }
}
```

```
///////////////////////////////////////////////////////////////////////
class WizardFactory : DComObject, IClassFactory
{
   override HRESULT QueryInterface(in IID* riid, void** pvObject)
   {
      if(queryInterface2!(IClassFactory) (this, IID_IClassFactory, riid, pvObject))
         return S_OK;
      return super.QueryInterface(riid, pvObject);
   }

   override HRESULT CreateInstance(IUnknown UnkOuter, in IID* riid, void** pvObject)
   {
      logCall("%s.CreateInstance(riid=%s)", this, _toLog(riid));

      assert(!UnkOuter);
      auto wiz = newCom!ItemWizard;
      return wiz.QueryInterface(riid, pvObject);
   }

   override HRESULT LockServer(in BOOL fLock)
   {
      if(fLock)
         InterlockedIncrement(&g_dllRefCount);
      else
         InterlockedDecrement(&g_dllRefCount);
      return S_OK;
   }

   int lockCount;
}

class ItemWizard : DisposingDispatchObject, dte.IDTWizard
{
   override HRESULT QueryInterface(in IID* riid, void** pvObject)
   {
      if(queryInterface!(dte.IDTWizard) (this, riid, pvObject))
         return S_OK;
      return super.QueryInterface(riid, pvObject);
   }
```

```
    override void Dispose()
    {
    }
    override ComTypeInfoHolder getTypeHolder ()
    {
        mixin(LogCallMix);
        return null;
    }

    override HRESULT Execute(/+[in]+/ IDispatch Application,
                             in int hwndOwner,
                             in SAFEARRAY* ContextParams,
                             in SAFEARRAY* CustomParams,
                             /+[in, out]+/ dte.wizardResult* retval)
    {
        mixin(LogCallMix);

        SAFEARRAY* sa = *cast(SAFEARRAY**)ContextParams;
        assert(SafeArrayGetDim(sa) == 1);
        LONG lbound, ubound;
        SafeArrayGetLBound(sa, 1, &lbound);
        SafeArrayGetUBound(sa, 1, &ubound);
        size_t cnt = (ubound - lbound + 1);

        string WizardType, ProjectName, /*ProjectItems,*/ LocalDirectory, ItemName,
InstallationDirectory;
        bool silent;

        VARTYPE vt;
        SafeArrayGetVartype(sa, &vt);
        if(vt == VT_VARIANT)
        {
            VARIANT var;
            LONG idx = lbound;
            if(SafeArrayGetElement(sa, &idx, &var) == S_OK && var.vt == VT_BSTR)
                WizardType = to_string(var.bstrVal);
            if(SafeArrayGetElement(sa, &++idx, &var) == S_OK && var.vt == VT_BSTR)
                ProjectName = to_string(var.bstrVal);
            ++idx;
```

```
        if(SafeArrayGetElement(sa, &++idx, &var) == S_OK && var.vt == VT_BSTR)
            LocalDirectory = to_string(var.bstrVal);
        if(SafeArrayGetElement(sa, &++idx, &var) == S_OK && var.vt == VT_BSTR)
            ItemName = to_string(var.bstrVal);
        if(SafeArrayGetElement(sa, &++idx, &var) == S_OK && var.vt == VT_BSTR)
            InstallationDirectory = to_string(var.bstrVal);
        if(SafeArrayGetElement(sa, &++idx, &var) == S_OK && var.vt == VT_BOOL)
            silent = var.boolVal != 0;
    }

    UtilMessageBox("Sorry, it does not make sense to add a package without specifying a
folder.\n" ~
                   "Please use the \"Add new item\" command from the project context menu.",
                   MB_OK, "Visual D - Add package");
    if(retval)
        *retval = dte.wizardResultCancel;
    return S_OK;
  }
}
```