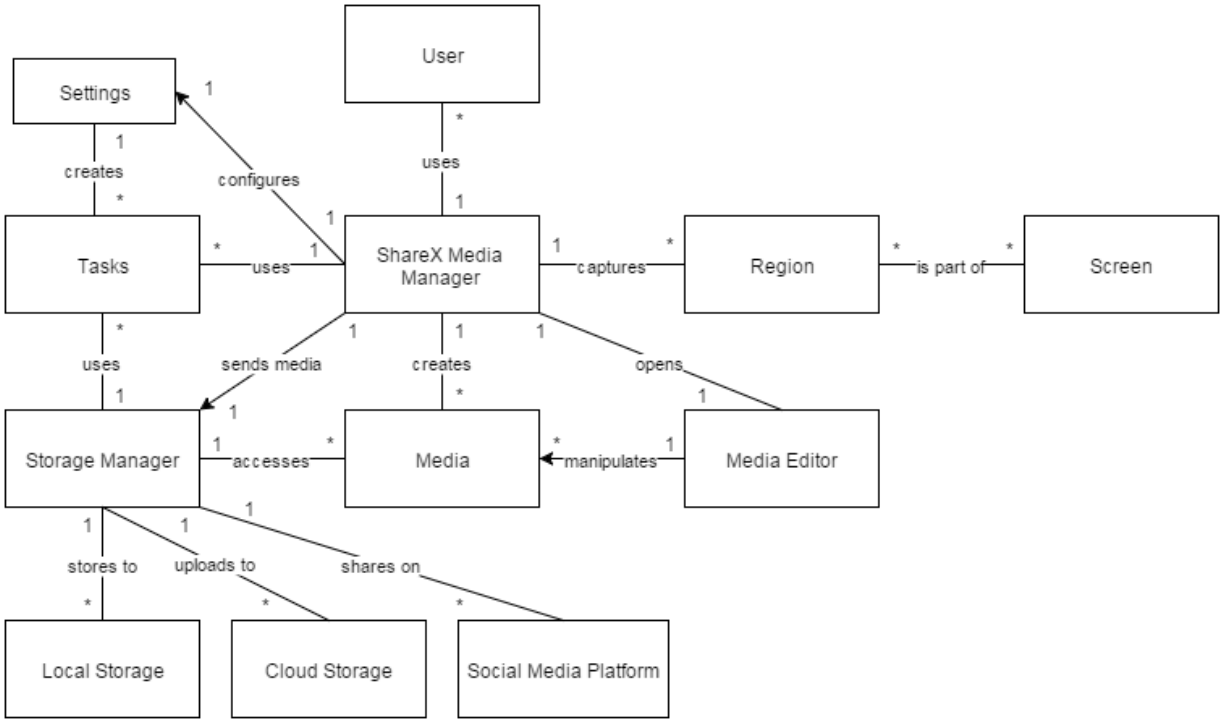
ShareX

David Bastien (26948553)
Hilary Chan (26984657)
Harley McPhee (27003226)
Lutherson Mendes (26746055)
Tom Mendakiewicz (26986099)
Roberto Ruffolo-Benavides (26974732)
Christiano Bianchet (27039573)

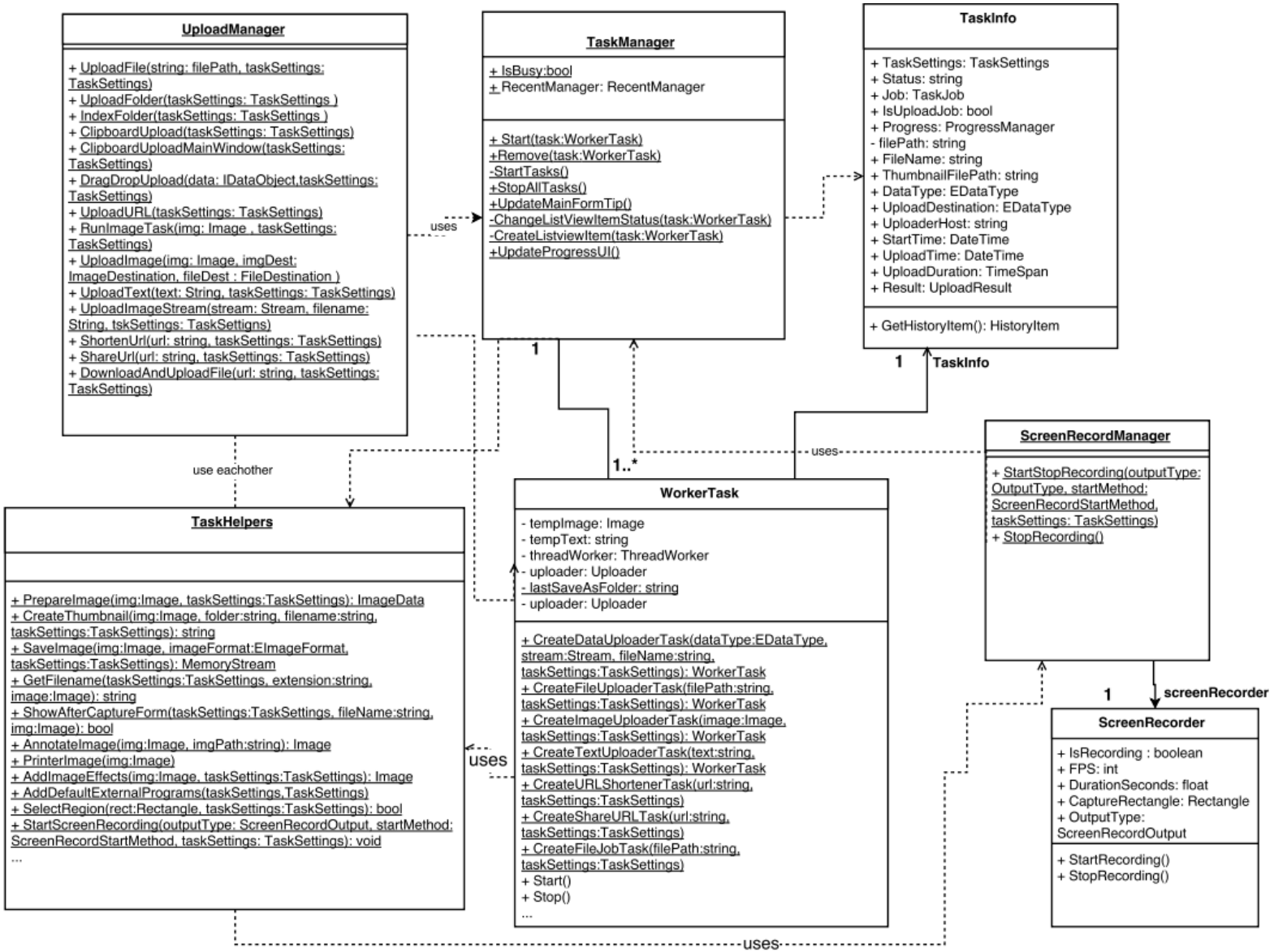
Github: https://github.com/lucson312/soen343/tree/M3

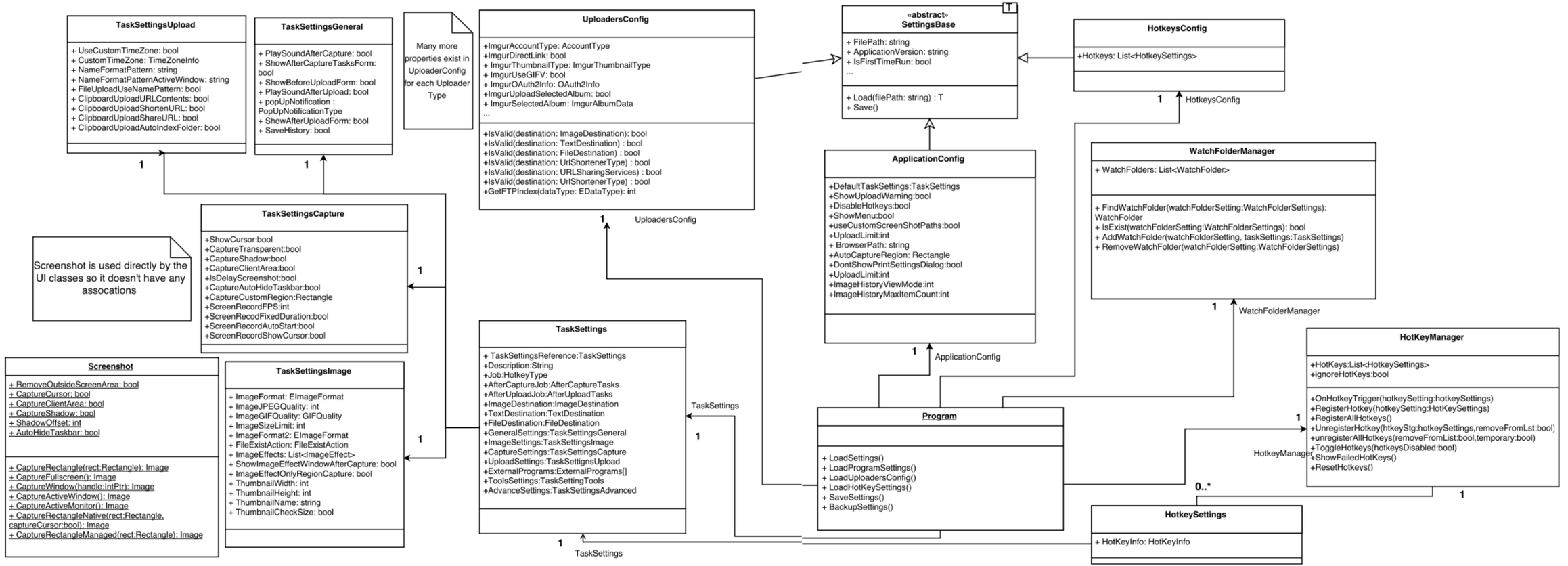
Summary of Project

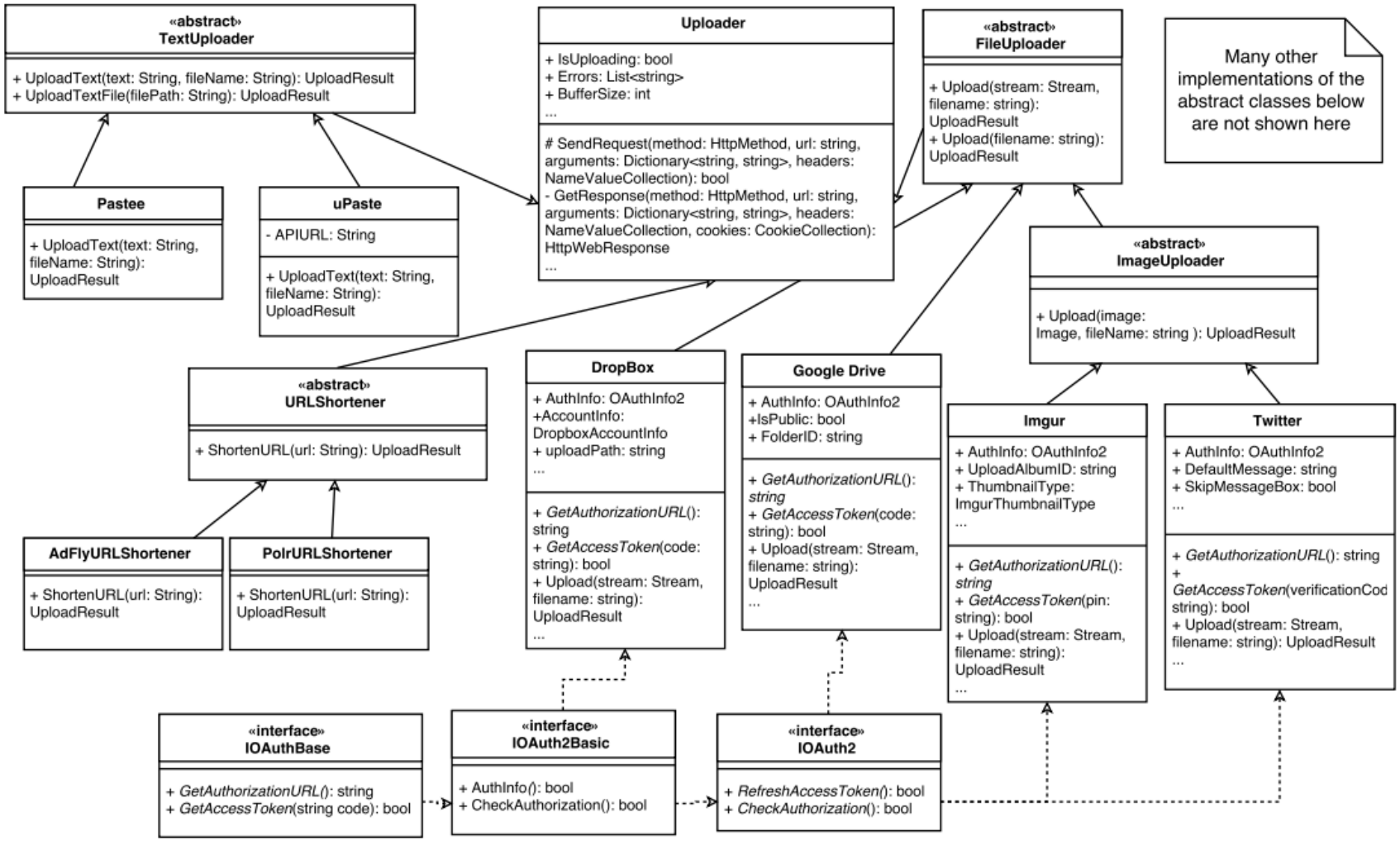
ShareX is a screen capturing software tool and a tool used to share text, images, and many other types of files. It has a lot of advanced features that most default operating system screen capturing applications lack. The reason we chose this project is because it is a relatively simple application to use and understand. The rich feature set adds a lot of depth to the project which will give us substance when we need to design the architecture of the actual application because a simple screen capturing application would hardly be enough. It also interests us as software engineering students as we will learn how to work with image files and upload them to various social media sites.Conceptual Diagram



Architecture Diagram (See link for full diagram image)


UploadManager
+ UploadFile(string: filePath, taskSettings: TaskSettings)
+ UploadFolder(taskSettings: TaskSettings)
+ IndexFolder(taskSettings: TaskSettings)
+ ClipboardUpload(taskSettings: TaskSettings)
+ ClipboardUploadMainWindow(taskSettings: TaskSettings)
+ DragDropUpload(data: IDataObject,taskSettings: TaskSettings)
+ UploadURL(taskSettings: TaskSettings)
+ RunImageTask(img: Image , taskSettings: TaskSettings)
+ UploadImage(img: Image, imgDest: ImageDestination, fileDest : FileDestination)
+ UploadText(text: String, taskSettings: TaskSettings)
+ UploadImageStream(stream: Stream, filename: String, tskSettings: TaskSettigns)
+ ShortenUrl(url: string, taskSettings: TaskSettings)
+ ShareUrl(url: string, taskSettings: TaskSettings)
+ DownloadAndUploadFile(url: string, taskSettings: TaskSettings)
uses
TaskManager
+ IsBusy:bool
+ RecentManager: RecentManager
+ Start(task:WorkerTask)
+Remove(task:WorkerTask)
-StartTasks()
+StopAllTasks()
+UpdateMainFormTip()
-ChangeListViewItemStatus(task:WorkerTask)
-CreateListviewItem(task:WorkerTask)
+UpdateProgressUI()
TaskInfo
+ TaskSettings: TaskSettings
+ Status: string
+ Job: TaskJob
+ IsUploadJob: bool
+ Progress: ProgressManager
- filePath: string
+ FileName: string
+ ThumbnailFilePath: string
+ DataType: EDataType
+ UploadDestination: EDataType
+ UploaderHost: string
+ StartTime: DateTime
+ UploadTime: DateTime
+ UploadDuration: TimeSpan
+ Result: UploadResult
+ GetHistoryItem(): HistoryItem
1
1
TaskInfo
use eachother
1..*
uses
ScreenRecordManager
+ StartStopRecording(outputType: OutputType, startMethod: ScreenRecordStartMethod, taskSettings: TaskSettings)
+ StopRecording()
TaskHelpers
+ PrepareImage(img:Image, taskSettings:TaskSettings): ImageData
+ CreateThumbnail(img:Image, folder:string, filename:string, taskSettings:TaskSettings): string
+ SaveImage(img:Image, imageFormat:EImageFormat, taskSettings:TaskSettings): MemoryStream
+ GetFilename(taskSettings:TaskSettings, extension:string, image:Image): string
+ ShowAfterCaptureForm(taskSettings:TaskSettings, fileName:string, img:Image): bool
+ AnnotateImage(img:Image, imgPath:string): Image
+ PrinterImage(img:Image)
+ AddImageEffects(img:Image, taskSettings:TaskSettings): Image
+ AddDefaultExternalPrograms(taskSettings,TaskSettings)
+ SelectRegion(rect:Rectangle, taskSettings:TaskSettings): bool
+ StartScreenRecording(outputType: ScreenRecordOutput, startMethod: ScreenRecordStartMethod, taskSettings: TaskSettings): void
...
WorkerTask
- tempImage: Image
- tempText: string
- threadWorker: ThreadWorker
- uploader: Uploader
- lastSaveAsFolder: string
- uploader: Uploader
+ CreateDataUploaderTask(dataType:EDataType, stream:Stream, fileName:string, taskSettings:TaskSettings): WorkerTask
+ CreateFileUploaderTask(filePath:string, taskSettings:TaskSettings): WorkerTask
+ CreateImageUploaderTask(image:Image, taskSettings:TaskSettings): WorkerTask
+ CreateTextUploaderTask(text:string, taskSettings:TaskSettings): WorkerTask
+ CreateURLShortenerTask(url:string, taskSettings:TaskSettings)
+ CreateShareURLTask(url:string, taskSettings:TaskSettings)
+ CreateFileJobTask(filePath:string, taskSettings:TaskSettings)
+ Start()
+ Stop()
...
uses
1
screenRecorder
ScreenRecorder
+ IsRecording : boolean
+ FPS: int
+ DurationSeconds: float
+ CaptureRectangle: Rectangle
+ OutputType: ScreenRecordOutput
+ StartRecording()
+ StopRecording()
uses


TaskSettingsUpload
+ UseCustomTimeZone: bool
+ CustomTimeZone: TimeZoneInfo
+ NameFormatPattern: string
+ NameFormatPatternActiveWindow: string
+ FileUploadUseNamePattern: bool
+ ClipboardUploadURLContents: bool
+ ClipboardUploadShortenURL: bool
+ ClipboardUploadShareURL: bool
+ ClipboardUploadAutoIndexFolder: bool
TaskSettingsGeneral
+ PlaySoundAfterCapture: bool
+ ShowAfterCaptureTasksForm: bool
+ ShowBeforeUploadForm: bool
+ PlaySoundAfterUpload: bool
+ popUpNotification : PopUpNotificationType
+ ShowAfterUploadForm: bool
+ SaveHistory: bool
Many more properties exist in UploaderConfig for each Uploader Type
UploadersConfig
+ImgurAccountType: AccountType
+ImgurDirectLink: bool
+ ImgurThumbnailType: ImgurThumbnailType
+ ImgurUseGIFV: bool
+ ImgurOAuth2Info: OAuth2Info
+ImgurUploadSelectedAlbum: bool
+ ImgurSelectedAlbum: ImgurAlbumData
...
+IsValid(destination: ImageDestination): bool
+IsValid(destination: TextDestination) : bool
+IsValid(destination: FileDestination) : bool
+IsValid(destination: UrlShortenerType) : bool
+IsValid(destination: URLSharingServices) : bool
+IsValid(destination: UrlShortenerType) : bool
+GetFTPIndex(dataType: EDataType): int
1
UploadersConfig
«abstract»
SettingsBase
T
+ FilePath: string
+ ApplicationVersion: string
+ IsFirstTimeRun: bool
...
+ Load(filePath: string) : T
+ Save()
HotkeysConfig
+Hotkeys: List<HotkeySettings>
1
HotkeysConfig
TaskSettingsCapture
+ShowCursor:bool
+CaptureTransparent:bool
+CaptureShadow:bool
+CaptureClientArea:bool
+IsDelayScreenshot:bool
+CaptureAutoHideTaskbar:bool
+CaptureCustomRegion:Rectangle
+ScreenRecordFPS:int
+ScreenRecodFixedDuration:bool
+ScreenRecordAutoStart:bool
+ScreenRecordShowCursor:bool
1
Screenshot is used directly by the UI classes so it doesn't have any assocations
ApplicationConfig
+DefaultTaskSettings:TaskSettings
+ShowUploadWarning:bool
+DisableHotkeys:bool
+ShowMenu:bool
+useCustomScreenShotPaths:bool
+UploadLimit:int
+ BrowserPath: string
+AutoCaptureRegion: Rectangle
+DontShowPrintSettingsDialog:bool
+UploadLimit:int
+ImageHistoryViewMode:int
+ImageHistoryMaxItemCount:int
1
ApplicationConfig
WatchFolderManager
+ WatchFolders: List<WatchFolder>
+ FindWatchFolder(watchFolderSetting:WatchFolderSettings): WatchFolder
+ IsExist(watchFolderSetting:WatchFolderSettings): bool
+ AddWatchFolder(watchFolderSetting, taskSettings:TaskSettings)
+ RemoveWatchFolder(watchFolderSetting:WatchFolderSettings)
1
WatchFolderManager
Screenshot
+ RemoveOutsideScreenArea: bool
+ CaptureCursor: bool
+ CaptureClientArea: bool
+ CaptureShadow: bool
+ ShadowOffset: int
+ AutoHideTaskbar: bool
+ CaptureRectangle(rect:Rectangle): Image
+ CaptureFullscreen(): Image
+ CaptureWindow(handle:IntPtr): Image
+ CaptureActiveWindow(): Image
+ CaptureActiveMonitor(): Image
+ CaptureRectangleNative(rect:Rectangle, captureCursor:bool): Image
+ CaptureRectangleManaged(rect:Rectangle): Image
TaskSettingsImage
+ ImageFormat: EImageFormat
+ ImageJPEGQuality: int
+ ImageGIFQuality: GIFQuality
+ ImageSizeLimit: int
+ ImageFormat2: EImageFormat
+ FileExistAction: FileExistAction
+ ImageEffects: List<ImageEffect>
+ ShowImageEffectWindowAfterCapture: bool
+ ImageEffectOnlyRegionCapture: bool
+ ThumbnailWidth: int
+ ThumbnailHeight: int
+ ThumbnailName: string
+ ThumbnailCheckSize: bool
1
TaskSettings
+ TaskSettingsReference:TaskSettings
+Description:String
+Job:HotkeyType
+AfterCaptureJob:AfterCaptureTasks
+AfterUploadJob:AfterUploadTasks
+ImageDestination:ImageDestination
+TextDestination:TextDestination
+FileDestination:FileDestination
+GeneralSettings:TaskSettingsGeneral
+ImageSettings:TaskSettingsImage
+CaptureSettings:TaskSettingsCapture
+UploadSettings:TaskSettingsUpload
+ExternalPrograms:ExternalPrograms[]
+ToolsSettings:TaskSettingTools
+AdvanceSettings:TaskSettingsAdvanced
1
TaskSettings
TaskSettings
1
Program
+ LoadSettings()
+ LoadProgramSettings()
+ LoadUploadersConfig()
+ LoadHotKeySettings()
+ SaveSettings()
+ BackupSettings()
HotKeyManager
+HotKeys:List<HotkeySettings>
+ignoreHotKeys:bool
+OnHotkeyTrigger(hotkeySetting:hotkeySettings)
+RegisterHotkey(hotkeySetting:HotKeySettings)
+RegisterAllHotkeys()
+UnregisterHotkey(htkeyStg:hotkeySettings,removeFromLst:bool)
+unregisterAllHotkeys(removeFromList:bool,temporary:bool)
+ToggleHotkeys(hotkeysDisabled:bool)
+ShowFailedHotKeys()
+ResetHotkeys()
1
HotkeyManager
0..*
1
HotkeySettings
+ HotKeyInfo: HotKeyInfo


«abstract»
TextUploader
+ UploadText(text: String, fileName: String): UploadResult
+ UploadTextFile(filePath: String): UploadResult
Uploader
+ IsUploading: bool
+ Errors: List<string>
+ BufferSize: int
...
SendRequest(method: HttpMethod, url: string, arguments: Dictionary<string, string>, headers: NameValueCollection): bool
- GetResponse(method: HttpMethod, url: string, arguments: Dictionary<string, string>, headers: NameValueCollection, cookies: CookieCollection): HttpWebResponse
...
«abstract»
FileUploader
+ Upload(stream: Stream, filename: string): UploadResult
+ Upload(filename: string): UploadResult
Many other implementations of the abstract classes below are not shown here
Pastee
+ UploadText(text: String, fileName: String): UploadResult
uPaste
- APIURL: String
+ UploadText(text: String, fileName: String): UploadResult
«abstract»
ImageUploader
+ Upload(image: Image, fileName: string): UploadResult
«abstract»
URLShortener
+ ShortenURL(url: String): UploadResult
DropBox
+ AuthInfo: OAuthInfo2
+AccountInfo: DropboxAccountInfo
+ uploadPath: string
...
+ GetAuthorizationURL(): string
+ GetAccessToken(code: string): bool
+ Upload(stream: Stream, filename: string): UploadResult
...
Google Drive
+ AuthInfo: OAuthInfo2
+IsPublic: bool
+ FolderID: string
+ GetAuthorizationURL(): string
+ GetAccessToken(code: string): bool
+ Upload(stream: Stream, filename: string): UploadResult
...
Imgur
+ AuthInfo: OAuthInfo2
+ UploadAlbumID: string
+ ThumbnailType: ImgurThumbnailType
...
+ GetAuthorizationURL(): string
+ GetAccessToken(pin: string): bool
+ Upload(stream: Stream, filename: string): UploadResult
...
Twitter
+ AuthInfo: OAuthInfo2
+ DefaultMessage: string
+ SkipMessageBox: bool
...
+ GetAuthorizationURL(): string
+ GetAccessToken(verificationCod string): bool
+ Upload(stream: Stream, filename: string): UploadResult
...
AdFlyURLShortener
+ ShortenURL(url: String): UploadResult
PolrURLShortener
+ ShortenURL(url: String): UploadResult
«interface»
IOAuthBase
+ GetAuthorizationURL(): string
+ GetAccessToken(string code): bool
«interface»
IOAuth2Basic
+ AuthInfo(): bool
+ CheckAuthorization(): bool
«interface»
IOAuth2
+ RefreshAccessToken(): bool
+ CheckAuthorization(): bool

The class Program is the main class that is used to load and contain all the users' settings for ShareX. The Program class has an association with various sub-classes of SettingBase such as: UploadersConfig, HotKeysConfig, and ApplicationConfig. Each one holds a specific configuration that ShareX will need in order to function. The classes mentioned also extend the abstract class SettingsBase. The class SettingsBase allows these subclasses to load and save the configuration that the user has specified, allowing for memory persistence between ShareX sessions.

Program also contains an instance of WatchFolderManager. WatchFolderManager allows ShareX to monitor added folders for any new files placed within them. Once a new file is placed inside this WatchedFolder, the corresponding TaskSettings in WatchedFolder will be performed on that file. Program also contains an instance of HotKeyManager which manages ShareX's hotkeys configuration so they can be changed, removed, or added. The last association the Program class has is with TaskSettings.

TaskSettings allows ShareX's other classes to know how to perform their specific functionality and what must be performed. For this reason, a majority of the classes have a dependency of some sort on the class TaskSettings. TaskSettings can allow for a file to be uploaded to various cloud storage sites or be copied to another destination. The class also contains one instance of many other classes that are used for settings such as: TaskSettingUpload, TaskSettingsGeneral, TaskSettingsCapture, and TaskSettingsImage.

Screenshot is a static class that is used by UI classes to perform various types of screen captures, which are listed in the enum CaptureType. The class doesn't interact or have any other associations with any other classes of the system. The class is used to obtain an Image by capturing the screen in some way. The UI classes will then perform AfterCaptureTasks. The AfterCaptureTasks can perform the tasks specified by the AfterCaptureTasks enum. If one of the AfterCaptureTasks specified that the image will be uploaded, then the AfterUploadTasks will be ran. AfterUploadTasks performs tasks such as: shortening the url of the uploaded image, sharing the url via social media, copying the uploaded image url to the clipboard or opening the url in the user's default browser.

Uploader is a class that provides common functionality to its subclasses. Three abstract classes extend Uploader: URLShortener, FileUploader, and ImageUploader. These abstract classes further provide various subclasses with common functionality that they all support, which allows the WorkerTask to fulfill its tasks. Some subclasses require verification and authorization in-order to interact with external systems, such as: DropBox, Google Drive, Imgur, and Twitter. These classes implement an authorization class: IOAuthBase, IOAuth2Baisc, or IOAuth2.

The static classes UploadManager and TaskManager are both used heavily by the UI classes. UploadManager allows for images, files, and text to be uploaded to various destinations. TaskManager can shorten urls and record the user's screen. These classes perform this functionality by creating an instance of WorkerTask which is used to execute some task in an asynchronous manner. The WorkerTask object gets information from the TaskSettings, which is then stored into TaskInfo from UploadManager. TaskInfo informs the WorkerTask of the tasks it must perform. Lastly, the WorkerTask will use the static class TaskHelpers to perform AfterCaptureTasks, and to record the user's screen.

One of the main discrepancies between the conceptual diagram and the actual class diagram is the importance placed on the TaskManager. The conceptual diagram had a ShareX Media Manager class as the center of the functionality, which interacted with the various managers and Media. In reality, this is not the case, as there is no central point of functionality. Instead, the various managers use the TaskManager and WorkerTask to function. For instance, the conceptual class MediaEditor is handled by

the actual class AfterCaptureTasks and is only one of its many functionalities as opposed to being a class of its own.

The conceptual diagram included region and screen classes which actually just fall in the CaptureType enumeration. The actual class diagram places more importance on AfterCaptureTasks and TaskSettingsCapture than the actual region being captured.

There are discrepancies between our conceptual diagram and ShareX's class diagram because we developed a conceptual diagram of an already functional application, whereas the ShareX developers only had a vision of what they wanted to develop. The architecture we created therefore is a lot simpler whereas the ShareX architecture is more complicated as a result of starting from the minimal functional system and adding new functionalities. The conceptual diagram allowed for easier legibility and understanding of the fundamentals of ShareX. On the other hand, the system's actual architecture is not as easy to follow as there are associations between many classes due to the center functionality being on the various task classes. Due to a weaker architecture implemented by ShareX, many classes have high coupling as a result making changes to the system may be cumbersome.

The reverse engineering tools used to create ShareX system class diagram was various features in Visual Studio. Code mapping was used which is a features of Visual Studio that generates a visualization of the interaction between classes in the system, it shows specifically what methods and property a class uses on other classes so it shows the coupling between classes and why they're coupled. Visual Studio also supports the generation of basic Class diagram without associations. Visual Studio also made it very easy to navigate through the large amount of classes by using features such as finding all references of a particular class, going to the class definition, and finding all subclasses of specific class.

Below is some of the code for the class TaskSettings, the class has associations with various other classes such as TaskSettingsCapture, TaskSettingsImage, TaskSettingsGeneral, and TaskSettingsUpload.

```
public class TaskSettings
{
    [JsonIgnore]
    public TaskSettings TaskSettingsReference { get; private set; }
    public string Description = string.Empty;
    public HotkeyType Job = HotkeyType.None;
    public bool UseDefaultAfterCaptureJob = true;
    public AfterCaptureTasks AfterCaptureJob = AfterCaptureTasks.CopyImageToClipboard |
AfterCaptureTasks.SaveImageToFile | AfterCaptureTasks.UploadImageToHost;
    public bool UseDefaultAfterUploadJob = true;
    public AfterUploadTasks AfterUploadJob = AfterUploadTasks.CopyURLToClipboard;
    public bool UseDefaultDestinations = true;
    public ImageDestination ImageDestination = ImageDestination.Imgur;
    public FileDestination ImageFileDestination = FileDestination.Dropbox;
    ...
    public bool UseDefaultGeneralSettings = true;
    public TaskSettingsGeneral GeneralSettings = new TaskSettingsGeneral();
    public bool UseDefaultImageSettings = true;
    public TaskSettingsImage ImageSettings = new TaskSettingsImage();
    public bool UseDefaultCaptureSettings = true;
    public TaskSettingsCapture CaptureSettings = new TaskSettingsCapture();

    }
```

Below is some of the code of TaskSettingsImage which is one of the classes TaskSettings contains in-order to specify settings for images.

```
public class TaskSettingsImage
{
    #region Image / General
    public EImageFormat ImageFormat = EImageFormat.PNG;
    public int ImageJPEGQuality = 90;
    public GIFQuality ImageGIFQuality = GIFQuality.Default;
    public int ImageSizeLimit = 1024;
    public EImageFormat ImageFormat2 = EImageFormat.JPEG;
    public FileExistAction FileExistAction = FileExistAction.Ask;
    #endregion Image / General
    #region Image / Effects
    [JsonProperty(ItemTypeNameHandling = TypeNameHandling.Auto)]
    public List<ImageEffect> ImageEffects = ImageEffectManager.GetDefaultImageEffects();
    public bool ShowImageEffectsWindowAfterCapture = false;
    public bool ImageEffectOnlyRegionCapture = false;
    #endregion Image / Effects
    #region Image / Thumbnail
    public int ThumbnailWidth = 200;
    ...
}
```

Code Smells and System Level Refactorings

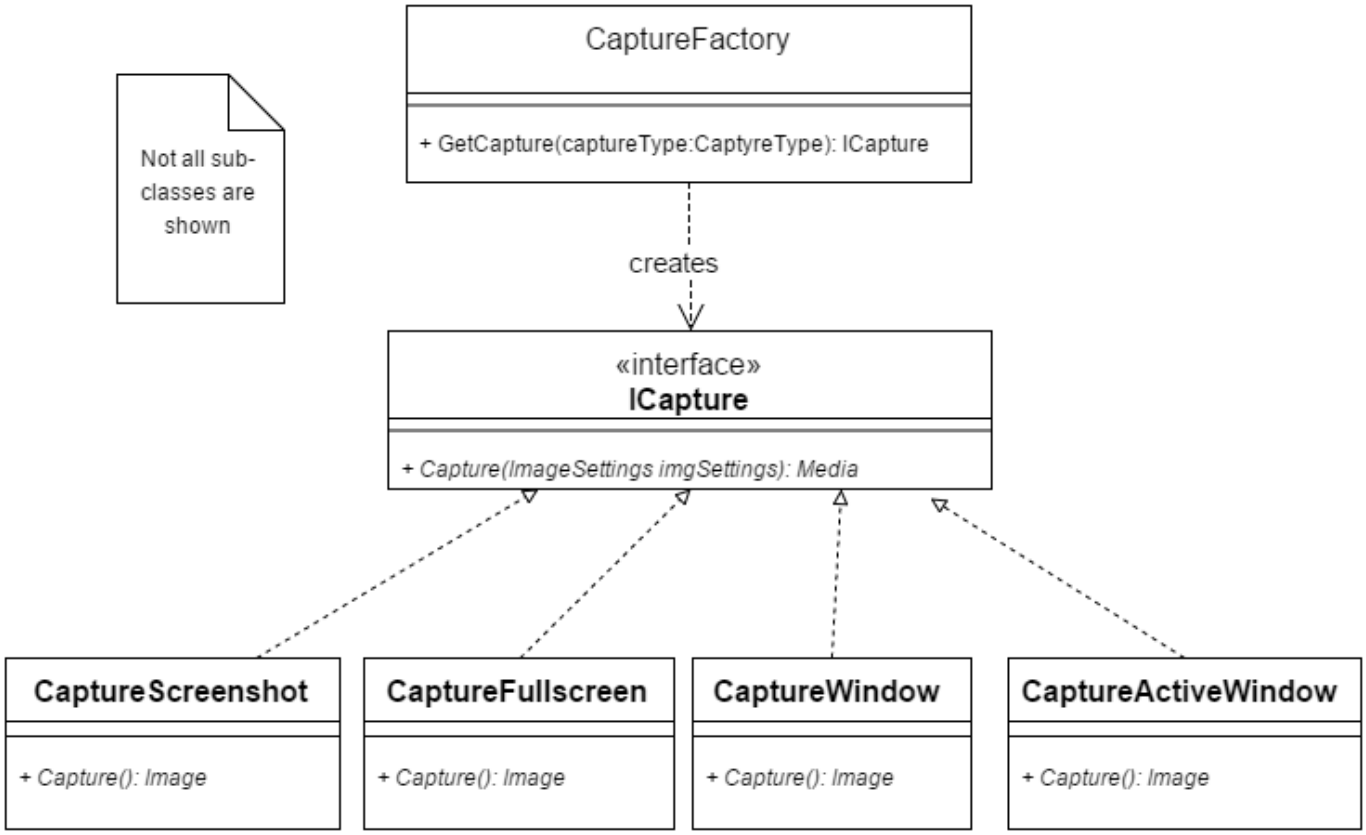
The God class MainFrom.cs contains too much domain logic that should be delegated to specific domain level classes and not a class such as MainForm that is used to primarily handle input events and to trigger domain logic on domain classes. Since it is a class that is mainly used for handling input events from the user interface, it doesn't need to know about domain logic implementation. Although currently it does delegate most of the tasks to domain classes, there is still a good amount of logic inside the classes that can be considered domain level logic.

This causes the class to become highly coupled, bloated and hard to read. For instance, the method CaptureScreenshot which handles the different types of screen capture, implements a switch and case operator to call different methods based on user's screenshots type at run time, this ideally should be hidden inside the domain logic classes using the factory and strategy pattern. The factory classes are the classes that would return the appropriate class object that are needed to perform the action that the user is requesting. The class that is returned would be a class that implements a common interface that the MainForm knows how to use, so it can simply use the interface without having to know the underlying details of how to perform each action.

We will use the Extract Class refactoring to remove domain logic method from the god class MainForm.cs, the new classes will be CaptureFactory, as well as the ICapture interface and several sub-classes that will implement the interface to perform the logic to either upload or capture a screenshot. By using the strategy pattern, we can break down the methods of CaptureScreenshot into different sub-classes that will implement the ICapture interface. These sub-classes can be narrowed down to CaptureRectangle, CaptureFullscreen, CaptureWindow and, CaptureActiveWindow and many more.

Since the MainForm class is handling the GUI request as well as the creation of domain object, we can encapsulate the process by creating Factories, we can assign the responsibility of determining which action to do to the factory, and the logic to execute the actions will be implemented in the implementation of each individual interface of ICapture. These strategies will increase cohesion and lower coupling while decreasing eventual introduction of code smells, such as object-orientation abusers and bloaters like long method and large classes. In addition, the class Screenshot.cs can be safely removed and its methods can be implemented by the appropriate sub-class of the Strategy pattern implementation.

A diagram of the planned refactoring can be seen below:



The class ScreenRecordManager.cs contains the bloated method:

```
private static void StartRecording(ScreenRecordOutput outputType, TaskSettings
taskSettings, ScreenRecordStartMethod startMethod = ScreenRecordStartMethod.Region){
        ...
}
```

This method is extremely long. A method can be considered long if it has greater than 10 lines. The StartRecording method contains 238 lines of code. It is a method that performs numerous tasks to the point that it is very difficult to read and follow what everything does. Some of these tasks involve managing the output format, handling the image settings, such as image quality and location. To fix these problems, we can use the Extract Method and create different classes for each task with their related methods. For instance, the task of creating an output file can be extracted to its own class where everything that is related to saving the new file is handled. We can also use a combination of strategy and factory pattern to handle the different output format of the file. This will increase cohesion and encapsulate the process of file creation.

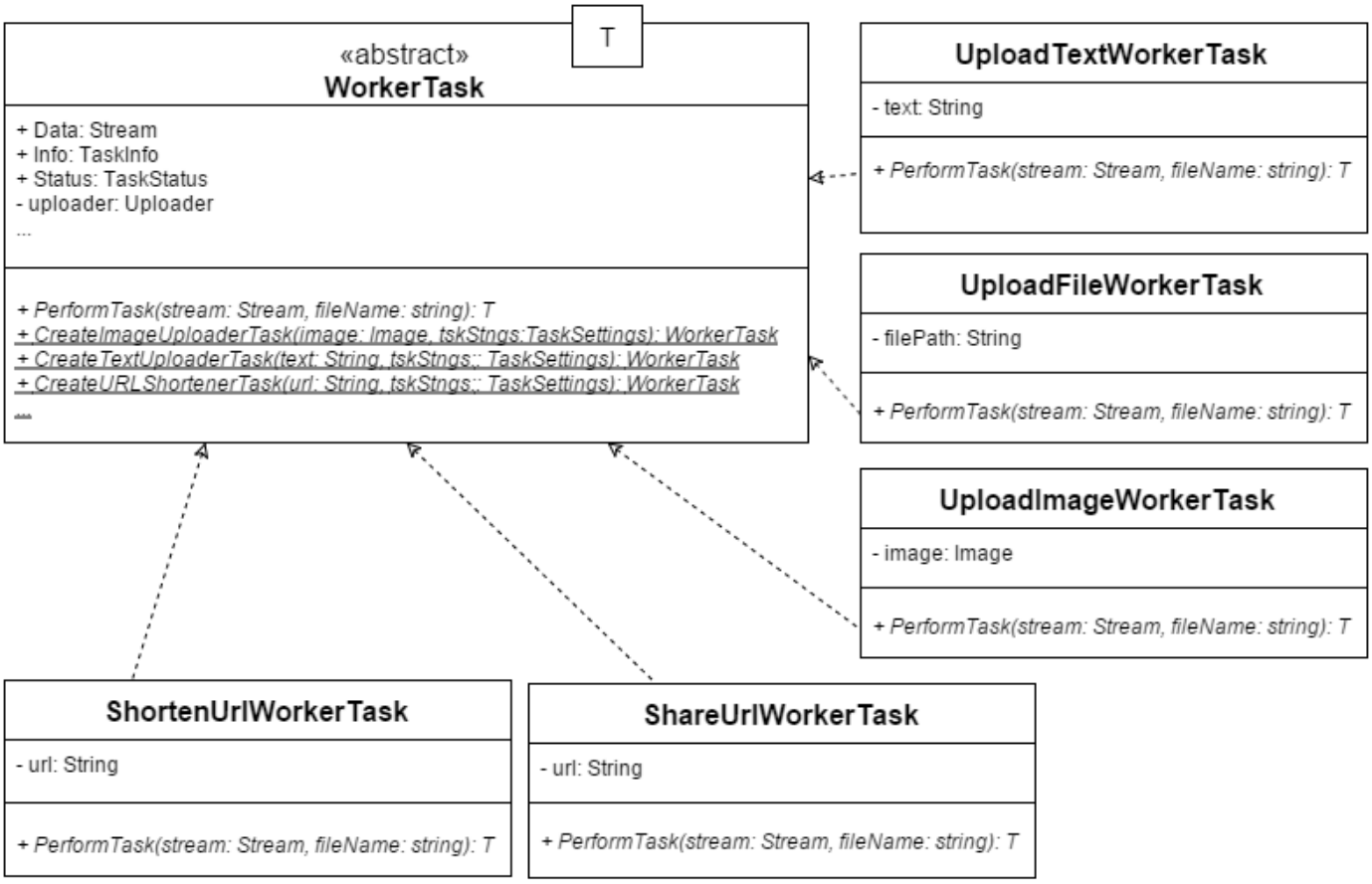
The method shown below, ClipboardUpload is a bloated method, with long chains of if-elses and operations performed in each segment. Due to this, the method becomes unreadable as the core functionality is obfuscated by many smaller operations that should have been extracted into new methods. The code denoted by ellipses also contains more else-ifs nested within the ones displayed. Extracting these segments of code into new methods will increase readability.

```
public static void ClipboardUpload(TaskSettings taskSettings = null)
{
	if (taskSettings == null) taskSettings = TaskSettings.GetDefaultTaskSettings();

	if (Clipboard.ContainsImage())
	{
		…
	}
	else if (Clipboard.ContainsFileDropList())
	{
		…
	}
	else if (Clipboard.ContainsText())
	{
		…
	}
}
```

The class WorkerTask is another class that has a few code smells such as long methods, large class, and it also has a significant amount of switch statements. The class is capable of doing too much, it has methods to UploadImages, UploadText, UploadFile, ShortrenUrl, and ShareUrl and each of these methods contain large amount of switch statements and other code to determine where to and to actually upload the image, text, or file to or how to shorten the url or share a url. The class can be refactored by breaking it down into many smaller classes and having a superclass that contains common functionality that each one will need, the corresponding subclasses could be UploadImageWorkerTask, UploadTextWorkerTask, UploadFileWorkerTask, ShortenUrlWorkerTask, ShareUrlWorkerTask. These subclasses will extend WorkerTask and use its methods and properties to determine what specific task it needs to do, for example UploadImageWorkerTask will contain the method UploadImage which will contain the switch statement to determine what instanceof uploader to use based on the images specified destination. Breaking WorkerClass up will help improve the class by making it smaller and contain fewer responsibilities. Another improvement that can be made is making WorkerTask an abstract class and specifying a virtual method such as PerformTask, and each task will be responsible for implementing their corresponding task in this method, so instead of having UploadImage in UploadImageWorkerTask it will be PerformTask and will contain the same logic, this is possible as all these methods have the same set of parameters and the use of generics can be used to specify the type that the method will return as various task return different types.

A diagram of the planned refactoring can be seen below



Specific Refactorings to be implemented in Milestone 4

As mentioned in the previous section, the main refactoring we will be doing in Milestone 4 involves the CaptureScreenshot method in the MainForm class. This method contains domain logic that does not belong in the MainForm class. It calls several related screenshot methods that could all be moved into multiple classes belonging to one common interface. Therefore, to remove the domain logic in the CaptureScreenshot method, we will first use the Extract Class refactoring method to move all domain logic from the MainForm class to separate classes. This domain logic will be refactored using the factory and strategy design patterns, dividing all the related screenshot methods into separate classes that have a common interface. An example of these methods that will be refactored are the different variations of the DoCapture method that can be seen in the screenshot below, which represent full screen captures and active monitor captures.

```
public void CaptureScreenshot(CaptureType captureType, TaskSettings taskSettings = null, bool autoHideForm = t
{
    if (taskSettings == null) taskSettings = TaskSettings.GetDefaultTaskSettings();

    switch (captureType)
    {
        case CaptureType.Screen:
            DoCapture(Screenshot.CaptureFullscreen, CaptureType.Screen, taskSettings, autoHideForm);
            break;
        case CaptureType.ActiveWindow:
            CaptureActiveWindow(taskSettings, autoHideForm);
            break;
        case CaptureType.ActiveMonitor:
            DoCapture(Screenshot.CaptureActiveMonitor, CaptureType.ActiveMonitor, taskSettings, autoHideForm);
            break;
        case CaptureType.Rectangle:
        case CaptureType.RectangleWindow:
        case CaptureType.Polygon:
        case CaptureType.Freehand:
            CaptureRegion(captureType, taskSettings, autoHideForm);
            break;
        case CaptureType.CustomRegion:
            CaptureCustomRegion(taskSettings, autoHideForm);
            break;
        case CaptureType.LastRegion:
            CaptureLastRegion(taskSettings, autoHideForm);
            break;
    }
}
```

To remove these methods (DoCapture, CaptureActiveWindow, CaptureRegion, etc.) from the MainForm class, we will move each of these methods to their respective class that implement the ICapture interface. For example, the CaptureRegion method will be moved to the RegionCapture class and will be renamed capture, like all the other methods that will be moved to their respective classes. After these methods have been moved to the ICapture classes, a CaptureFactory will be implemented so that the CaptureScreenshot method can retrieve the proper ICapture class without having to know the creation logic. It can then execute the capture method. Overall, this refactoring reduces coupling in the MainForm class, and raises cohesion.

Another necessary refactoring would be for the startRecording method in the ScreenRecordingManager class. This method contains a lot of code and could be divided up into smaller methods. To do this, we use the Extract refactoring method, allowing us to create classes for the different

tasks performed in the startRecording method. Therefore, tasks inside the startRecording method like the one that creates an output file for the recording will have their own class and methods. In startRecording, there is also the handling of the different output formats for the recording. Instead of having all of this inside the method, a combination of the factory and strategy pattern can be used to isolate the different output formats and the methods used to handle them.

The last refactoring that will be performed in the next milestone will be the class WorkerTask, this class contains many large methods which contributes to it being a very large and bloated class. It will be refactored by specifying the WorkerTask as an abstract class and specifying common operations that subclasses will implement by allowing methods to be removed from WorkerTask and implemented in these subclasses. This will allow WorkerTask to be have higher cohesion by reducing the size and capability of it.