Harnessing Collective Software Development

Luis Artola

Digital Domain
Venice, California, USA
la@luisartola.com

Abstract

Python has become established as the *de facto* scripting language in many industries including the post-production of visual effects. Its shallow learning curve enables a wider range of individuals to produce code much more quickly than before. This often leads to increased code duplication, competing tools and increased maintenance costs. This talk presents an attempt to harness all that coding power in a fast-paced production environment with the intention of increasing code reuse, reducing maintenance costs and improving the quality of the development process and the code itself. It describes philosophy, tools, techniques and challenges of harnessing collective software development in the pursuit of better software.

Categories and Subject Descriptors D.1.0 [*Programming Techniques*]: General; D.2.1 [*Software Engineering*]: Requirements / Specifications—Methodologies; D.2.2 [*Software Engineering*]: Design Tools and Techniques—Modules and interfaces; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Top-down programming; D.2.8 [*Software Engineering*]: Metrics—Process metrics; D.2.9 [*Software Engineering*]: Management—Cost estimation; D.2.13 [*Software Engineering*]: Reusable Software—Reuse models

General Terms Design, Documentation, Experimentation, Languages, Management, Measurement, Standardization.

Keywords Object-oriented Programming, Procedural Programming, Software Development Methologies, Software Process, Python.

1. Introduction

Collective Software Development takes place in environments where there are multiple simultaneous projects that require development of software tools. Each project has its own team of developers ready to quickly produce code to keep up with demaning schedules. The focus of a project is not the production of software *per se*, but support software is essential for the end result.

Each project has very similar needs, but time pressure tends to inhibit communication and code sharing amongst developers working on similar tools and scripts. This inevitably leads to code duplication and the proliferation of competing similar tools at the end of a few projects. The cost of maintenance and training increases as well. And, the amount of code that is eventually thrown away can be substantial.

Collective development is not the same as collaborative development. The former is normally done by a group that is not working in concert. The latter implies some level of communication and sharing.

The production of visual effects is a perfect example of such environments with an additional twist. Python is widely used in the industry to create supporting software and extend applications. Its ease of use enables people with little or no formal education in software engineering to produce code [4]. This usually results in code that is not easily reusable or maintainable, and is poorly documented.

This paper describes an approach to:

- Bridge the gap from collective to collaborative.
- Add structure to an otherwise disorganized development [6].
- Facilitate capturing and communicating knowledge across time and developers [3].
- Increase code reuse.

2. Methodology

2.1 Traditional scripting

Let's illustrate concepts and approach using an example.

The script in Figure 1 shows what a developer would typically write using only Python to solve a specific problem in production. The code performs a series of steps for taking some digital elements, validating, exporting different file formats, registering files in production tracking systems and cleaning up.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.

```
def publish():
  elements = getSelectedElements()
  validate()
  paths = getFilePaths()
  exportAnimation( elements, paths )
  exportGeometry( elements, paths )
  exportBoundingBoxes( elements, paths )
  exportThumbnails( elements, paths )
  registerFiles( paths )
  cleanUp()
```

Figure 1. publish.py: A traditional Python script.

There is nothing fundamentally wrong with this approach. But, the reality is that code is never written in a way that lends itself for easy reuse, especially not under time pressure. It is normally a highly cohesive mix of user interface, business logic and data transformation that is not easily modularized. This matters because at the same time a specific department in the production pipeline needs tasks like this to be coded and automated, there are people in other departments with similar needs.

2.2 Dividing and organizing code

This can be improved by following some basic principles to add structure and policies to the code:

1. Programs are **static descriptions** of what needs to happen, instead of actual code that only programmers can read and understand.
2. Every step in that program is **an individual module** that performs one and only one well defined operation, instead of a function in a larger piece of code [5].
3. **Data is separated from logic** and is planned, well-defined and predictable.

A program becomes a tree of connected modules that transform data [4, 5]. We refer to this as the *workflow* approach.

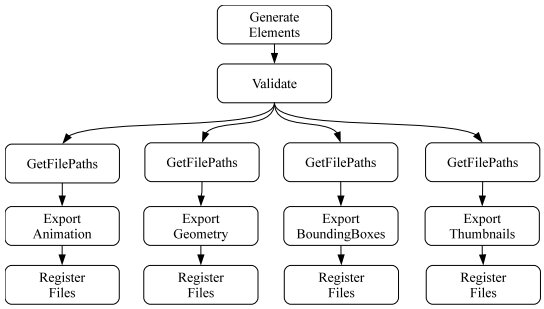


Figure 2. Workflow: A tree of connected modules

Figure 2 represents an alternative structure functionally equivalent to the script in Figure 1. The diagram shows the following elements:

Workflow Static description of connected actions.
The entire tree.

Process Sequence of actions in a workflow.
Each branch or subtree.

Action A discrete operation on application-specific data.
Each node.

```
<?xml version="1.0"?>
<workflow>
  <process name="main">
    <action module="GenerateElements"/>
    <action module="Validate"/>
    <process name="ExportAnimation">
      <action module="GetFilePaths"/>
      <action module="ExportAnimation"/>
      <action module="RegisterFiles"/>
    </process>
    <process name="ExportGeometry">
      <action module="GetFilePaths"/>
      <action module="ExportGeometry"/>
      <action module="RegisterFiles"/>
    </process>
    <process name="ExportBoundingBoxes">
      <action module="GetFilePaths"/>
      <action module="ExportBoundingBoxes"/>
      <action module="RegisterFiles"/>
    </process>
    <process name="ExportThumbnails">
      <action module="GetFilePaths"/>
      <action module="ExportThumbnails"/>
      <action module="RegisterFiles"/>
    </process>
  </process>
</workflow>
```

Figure 3. Publish.xml: Workflow equivalent of publish.py

```
class ExportAnimation( Action ):
  def run( self, data ):
    for element in data[ 'elements' ]:
      # export animation from element
    return data
```

Figure 4. ExportAnimation.py: Python module containing an action.

The workflow in Figure 2 is implemented with the following files:

- Workflow: An XML file that describes the operations to be performed (Figure 3.)
  - Publish.xml

- Actions: A Python module per operation that contains a single Action class. (Figure 4 shows an example action.)
  - GenerateElements.py
  - Validate.py
  - GetFilePaths.py
  - ExportAnimation.py
  - ExportGeometry.py
  - ExportBoundingBoxes.py
  - ExportThumbnails.py
  - RegisterFiles.py

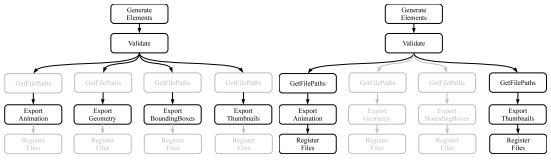
The actual separation of the initial Python script into multiple files can appear as overhead at first glance. However, the code that was originally written as functions inside a monolithic script is now packaged individually in smaller modules. The main function was replaced by an XML description. In terms of lines of code, the overhead is not significant [6]. And, the many benefits that will be explained shortly more than compensate for the added structure.

Even at this early stage, Figure 3 shows evidence of how smaller modules have now better chances of being reused. For example, the `GetFilePaths` and `RegisterFiles` actions are reused four times in the same workflow. And, since it is a standalone module with a very simple and well-defined interface, it has a higher probability of being reused in other workflows.

2.3 Static descriptions and XML

Having a workflow be a static description encoded in XML was somewhat controversial at the beginning. However, it provides many advantages, let's elaborate on some of them:

1. A static description is a contract, not only for the developer and ultimately the programming language, but also enables other, non-programmers and less techical stakeholders to get closer to understanding what a particular workflow in production does.



(a) Horizontal edits (b) Vertical edits

Figure 5. Workflow variations

2. Because the description is expressed in XML, making changes, adapting or quickly responding to changes in production due to external factors becomes *a configuration problem, not a programming task.* Examples include changes in the back-end, temporary disruption of certain services, etc. that should not require modifying the code. This is particularly valuable in fast-paced production environments where the response time is often times critical. Figure 5 shows variations of the workflow in Figure 2 to disable certain aspects of the workflow by simply removing select actions or entire processes.

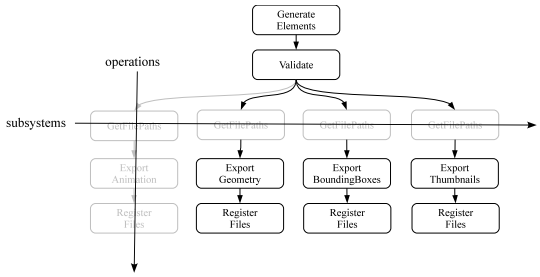


Figure 6. Workflow variations

3. An interesting property discovered after analyzing changes to workflows over time is that vertical edits normally translate to disabling the generation or transformation of certain data that was useful in some projects, but not in others. Horizontal edits normally translate in disabling or changing interaction with support systems. See Figure 6
4. Another important benefit of using XML is that the description of what a program would do becomes a service for a number of consumers. The primary consumer is of course the framework that executes the Python code that is referenced in the XML. But, because actions are very application-specific, XML enables the ability to gather statistics, extracting documentation from modules, etc. *without* having to actually run a live Python session to import modules. In the majority of cases an approach like this would be impractical as many actions would not have all the application-specific Python modules they need to even be imported in the first place.

2.4 Overriding code

Dividing code in workflows and actions is a hybrid of two programming approaches. Workflows are akin to procedural programming albeit at a higher level. Actions are still coded using object-oriented programming. One of the benefits of object-oriented programming is the ability to specialize code and override specific functionality to adapt to the needs of a specific project. Normally, this is accomplished using inheritance. And this is still true for individual actions at the implementation level. The question is how to provide a similar mechanism at the modular level used in the worklfow specification.

There is a simple solution. Use the file system to provide different locations where workflows and actions can be installed. Locations have precedence. When executing a workflow, actions are searched in those locations starting from the one that is *closer* to the user environment. To explain how this is determined, let's look at locations first. There can be four locations on disk for installing workflows and actions:

1. framework
2. shared
3. project
4. user

While it is possible to add an arbitrary number of levels, more than four are not practical in real world use. These levels provide automatic overriding for both workflows and actions. The framework that executes workflows reads the XML and locates actions based on the user environment. The *user* level is closer to the user, *framework* is farther way.

The level to use can be part of the environment, where the tools indicate what is the closest level for that particular execution.

This setup has some interesting properties:

The closer actions and workflows get to the *framework* level, the more stable they are. But, also the harder it is to make changes because they affect many other workflows.

Likewise, the closer they get to the *user* level, the less stable they are. However, the easier it is to make changes and do faster prototyping.

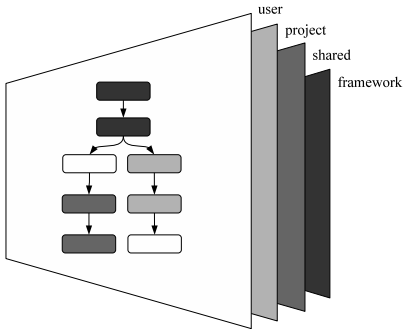


Figure 7. Overriding actions selectively

These levels can be visualized in Figure 7. Because actions and workflows can live at any given level, looking at a workflow definition is like looking through a series of translucent windows. A workflow is a projection of actions from different overriding levels.

Overriding is possible at a very granular level. This flexibility makes it also possible to quickly debug and diagnose problems at specific points in the process. For example, there are situations where a given workflow installed at the project level is failing in a particular action. A developer would simply copy that action to the *user* level and make changes, set breakpoints, etc. for easier troubleshooting.

2.5 Executing workflows

Traditional Python scripts control their internal data structures and can be executed directly from a command interpreter. Workflows on the other hand need a supporting framework for execution. The framework can be very nimble though, at the bare minimum it simply parses the XML file, locates and imports Python modules representing each individual action, and executes them one at a time sequentially passing data from one to another.

Actions are free to modify, append and destroy the data that is flowing through as needed. `Action` is the base class for all actions and has an extremely simplified interface. The only thing required is a method with the following signature: `def run( self, data )`.

Leveraging the highly-dynamic nature of Python when it comes to data types, `data` can be anything and can be transformed [4] as it flows down from action to action. Defining data structures that are simple to understand is one of the most challenging aspects of writing reusable actions. In our experience, all the workflows used a plain dictionary.

The following is a contrived example but representative of real use. Error checking and obvious import statements are intentionally omitted to illustrate how data flows downstream and is used and transformed by each action in turn.

```
<?xml version="1.0"?>
<workflow>
  <process name="main">
    <action module="GetElements"/>
    <action module="CreateDirectories"/>
    <action module="WriteContents"/>
  </process>
</workflow>
```

Figure 8. example.wam

```
class GetElements( Action ):
  def run( self, data ):
    data[ 'elements' ] = [
      dict( name='house' ),
      dict( name='car' ),
      dict( name='trees' ),
    ]
    return data
```

Figure 9. GetElements.py

```
def CreateDirectories( Action ):
  def run( self, data ):
    for element in data[ 'elements' ]:
      directory = '/elements/' + element.name
      element[ 'directory' ] = directory
      os.mkdir( directory )
    return data
```

Figure 10. CreateDirectories.py

```
class WriteContents( Action ):
  def run( self, data ):
    for element in data[ 'elements' ]:
      file_name = \
        element.directory + '/contents.txt'
      output = open( file_name, 'w' )
      output.write( element.name )
      output.close()
    return data
```

Figure 11. WriteContents.py

The workflow in Figure 8 illustrates how three actions work in concert to create files on disk. The `GetElements` action listed in Figure 9 gets executed first and seeds the `data` dictionary with three elements represented by simple dictionaries containing their names. In a real world scenario, an action like this would pull data from a database of some sort, e.g. an asset management or production tracking system.

The data returned by this action is then passed on to the `CreateDirectories` action listed in Figure 10. This action demonstrates how to use data coming into the action and modifying it for further use downstream. It uses the `name` key of each element in the `elements` list to build a directory path on disk. The directory is created and added as a `directory` key to each element dictionary.

The data is once more passed along to the third action listed in Figure 11, `WriteContents`. This action iterates over the `elements` using their `name` and `directory` keys to write a simple text file on disk for each element.

The beauty of separating data and logic this way is that other workflows can easily reuse, for example, the `CreateDirectories` action. The only thing there is to know about that action is that it requires a dictionary-like object to be passed in as the `data` argument. It also expects `data['elements']` to be a list of dictionary-like objects that provide access to a `name` key. Therefore, there exist implicit connections between modules based on the assumptions they make on each other [5].

This methodogy does not enforce any specific data structure to be used in actions. However, the simpler and better documented the data structures and the data that each individual action requires, the easier it would be to reuse them.

3. Metrics, Analysis and Benefits

3.1 Enabling collaboration

This methodology was used in multiple ocassions to scope out a particular problem to solve in production, devise a plan of attack, arrange and connect all the pieces of functionality that would need to be executed, divide the work and have multiple developers simultaneously code portions of the workflow much faster and with less interdependencies than traditional scripting.

Besides making better use of coding resources simultaneously, it also improves collaboration amongst developers in sucession over time. This is possible because workflows are modular by definition, every developer new to a workflow can contribute more easily because actions are very targeted.

To illustrate this point, a great practical example in our experience was the creation of a workflow to automate the preparation, packaging and delivery of digital assets and related data to outsourcing vendors. From a high level perspective, the problem required coding in the following domains:

- Filtering and coverting elements to make sure that no proprietary elements found their way outside the facility. This required a developer with intimate knowledge of the mayor artistic applications and tools used in production.
- Extracting SQL records from various back-end systems in a way that could be easily processed on another database cluster. This required a developer with database experience.
- Packaging, transferring and bookkeeping in various production tracking systems. This required a developer with knowledge of file system, online file distribution and production tracking systems.

It is easier to find three different developers with experience in one of these areas each than it is to find a single developer that masters them all. This is where enforcing modularity and separation of data from logic really stands out. In this particular case, the workflow was planned to be divided in three sections. Each developer added the necessary actions to the section they owned. This happened incrementally with no impact to other actions in the workflow.

The only thing that developers needed to agree on beforehand was the type and contents of the data that needed to flow from one action to another. In this example the data was decided to be a plain dictionary with the following keys:

- `elements`: A list of instances of an `Element` class that had well-known attributes to identify a digital asset tracked in our asset management systems. The absolute minimal was the unique ID.
- `source`: Name of the facility originating the package.
- `destination`: Name of the facility receiving the package.
- `location`: Full-path to the directory where all the actions in the workflow were to write relevant data related to the list of elements.

Besides having better utilization of coding resources and their expertise, it facilitated incremental development and more agile development and test iterations.

After this was implemented, the volume of data that was processed by this workflow grew very rapidly and perfor-

mance started to suffer. Profiling this code to gather timing statistics and narrowing down the bottleneck was a very straightforward approach if not trivial thanks to its modular implementation.

More importantly, with performance data in hand, it was possible to bring programmers and non-programmers alike to the table in the same room to analyze it in the context of the workflow description. It was discovered that packaging all the files from disk was the action that took the longest depending on the number of elements to process. The solution was to simply remove that one action from the workflow and creating a new workflow with just that one action to execute as a secondary step. This proved that fixing problems and adpating to unforseen issues can be solved as a *configuration* change, not a *programming* task.

3.2 Code reuse

One of the motivations for this methodology was code reuse. After a year of development applying it in various projects we measured how well it fared. The following table shows the number of workflows an actions written for various projects:

Level	Workflows	Actions
shared	94	216
project 1	340	560
project 2	158	264
project 3	94	127
project 4	108	209

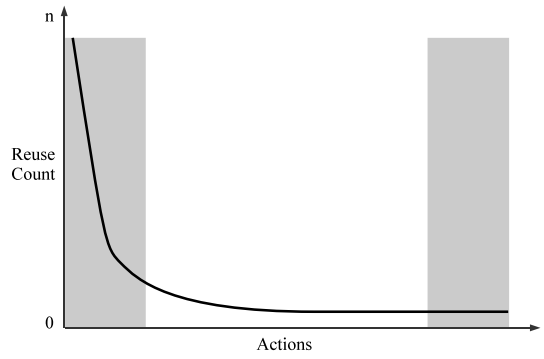


Figure 12. Action reuse pattern

Measuring code reuse required parsing workflows and gathering statistics about what actions were being reused, by which workflows and how often. After normalizing and graphing usage counts, the first intersting discovery was that every project exhibited exactly the same usage pattern: an inverse exponetial as illustrated in Figure 12.

Actions reused the most are on the left and reuse count decreases rapidly as we move on the horizontal axis towards the right. Further scrutiny revealed some interesting properties of this usage pattern. Reuse can be divided in three sections.

The left-most section is a range of high reuse counts. It is not uncommon to find actions reused over 50 times or even 100 times or more by many different workflows. Actions in this range are clearly reusable and should be protected and embraced as the official solution for the particular tasks they solve.

The middle section contains a range of reuse counts that is high enough to indicate that they have some value for a number of workflows, but not high enough to be considered widely useful for a variety of tasks. Actions in this range normally turned out to be very specific to a particular domain in production. One reason could be that the process was only applicable to a very particular artistic need in a project, not easily transferrable to others. Another reason was that such actions were a competing solution to similar other actions that turned out to be favored by other developers and yielded higher reuse - this phenomenon resembles natural selection in software development.

The right-most section is a range of extremely low reuse counts. Normally, there is a tail of actions that only achieve a reuse count of 1. The first impression is that such actions are probably just one-offs that could hardly be reused or not worth trying to. While that was true in some cases, another interesting discovering was that actions in this group turned out to be the one and only way of solving a very particular problem. These actions were prime candidates for protection to prevent competing solutions from sprouting to avoid duplicated efforts and wasted development resources.

Code reuse then is more easily identified with this methodology and it is characterized by those actions within the highest and the lowest reuse count ranges. Considering the lowest count as reusable becomes less counterintuitive by understanding that code reuse has two sides:

- Code that is reused many times by developers in other programs.
- Code that is written once and used many times by different users, as opposed to allowing competing solutions to appear over time instead of maintaining and extending the existing solution.

3.3 Code maturity vs. branching

Once the most reusable actions are identified, the challenge is making sure that they are protected and promoted to a higher level in the hierarchy of modular overriding. That is, moving actions vertically from the *project* level to *shared* and ultimately to the *framework* level. Code maturity can be measured by the transition of any given action to higher levels.

Code maturity is a worthy goal, but in practice, as new projects get started, copy-and-paste is by far the most common form of code reuse. Rather than having code percolate to higher levels, it replicates horizontally to the domain of other projects. Code branching happens on every copy leading eventually to multiple variations of the same action. Over

time, it becomes more difficult to keep track and reconcile code differences.

There are many reasons why branching is favored over maturity. It is easier to copy and adapt as needed than it is to spend the time and effort in cleaning up, generalizing and promoting existing code to higher levels so that they can be more widely reused. Time pressure is certainly the number one cause. Human aspects also play an important role. For example, the lack of interest in contributing to build good code legacy. Or, simply avoiding the responsibility inherent to making sure an action promoted to a higher level not only works for the new intended uses, but that it does not break existing workflows depending on it.

While this methodology facilitates code maturity and encourages it, it does not prevent code branching. However, it makes it more manageable. Because workflows can be parsed, it is easy to further analyze the actions they use and perform side by side comparisons with similar code from different projects.

3.4 Breeding grounds

This methodology is particulary suitable for breeding new ideas in faster code-test-retrofit cycles by leveraging the hierarchical module overriding levels. Complete applications can be coded in a modular way with a collection of workflows and actions. The maturity path normally goes from the *user* level to the *project* level. Once the feature set reaches a milestone, workflows can be parsed to locate all of the dependent actions and code can be physically carved out or copied from the different levels into self-contained packages.

3.5 Abstracting process execution.

A wide variety of domain-specific applications used in production come bundled with an embedded Python interpreter for providing scripting support. Applications for modeling, texturing, animation, effects, compositing, lighting, etc. It is very common for tools used in production to have to run certain portions of code within the context of a very specific interpreter. For example, the conversion of a geometry file from one format to another by scripting some commands of a modeling tool.

The `<process/>` tag can be extended with an attribute to indicate what kind of interpreter to use when executing the actions inside it. The framework can handle all the technicalities of executing a subprocess within the context of the given application. For example `<process interpreter="maya"/>`[1].

4. Tool set.

This methodology can be implemented with a simple framework that imposes minimal overhead, both in code footprint and execution time. A minimalistic implementation requires three components:

1. A Python API that provides simple classes that represent the action tree described in XML.
2. A custom Python module importer that can locate modules by name using relative paths that can be found paying attention to the mulitple overriding levels on disk.
3. Optionally, a command-line interface for executing worflows directly from a shell and other utilities.

4.1 Python API

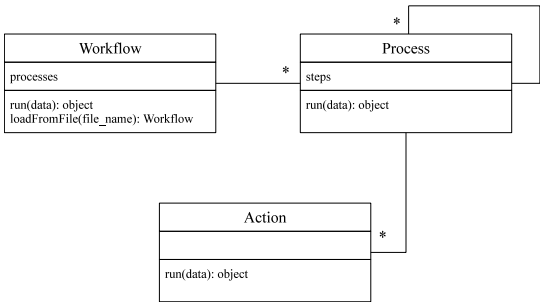


Figure 13. Minimal core components.

Figure 13 shows the minimal core classes required to represent and execute a tree of connected actions described in an XML workflow. `Workflow`, `Process` and `Action` provide an extremely simple interface to execute them by calling `run(data)`, passing along some data from step to step and returning the modified data.

`Workflow.loadFromFile` is a factory method that locates an XML on disk using the overriding levels on disk, parses it and constructs a tree of instances of the core classes. The actual modules that represent each action can be lazily imported and instantiated.

4.2 Module importer

A custom module importer is required for two main reasons.

The first one is that modules can be overriden using the various overriding levels described in section 2.4. The importer would append a relative path to all entries in the system path and try to locate and import the module.

Second, the nature of Python packages does not allow overriding subpackages and submodules of a package that exists in multiple overriding levels. Because folders in the file system are used broadly to classify actions per subsystem, the custom importer locates modules in different relative paths.

4.3 Optional command-line interface.

A very simple interface with commands and subcommands can be optionally written to execute workflows directly from a shell; copy all modules depending on a worflow, inspect, etc. The interface looks something like this:

[1] Maya is an application commonly used in the Visual Effects industry for various departments including modeling and animation.

- `workflow run`: Executes a workflow.
- `workflow stats`: Gathers statistics about action usage, code reuse, etc.
- `workflow diff`: Compares two workflows and their actions side by side.

5. Visualizing code development and evolution.

Every single module used by workflows is stored in its own file and tracked individually in a source control system. This presents very interesting opportunities to have a better insight on various aspects of software development for leaders, managers and software developers alike.

5.1 Statistics

All the action modules were broadly classified in folders named after the subsystem or application they were intended for, e.g. asset management, production tracking, Maya, Houdini[2], Nuke[3], utilities, etc. The following table shows statistics gathered from the Maya folder:

Metric	Value
First checkin	8/25/2009
Last checkin	11/09/2010
Active development time (months)	14.5
Number of files	103
Lines of code (LOC)	15,545
Developers	12
Average LOC per file	150
Average LOC per developer	1,295
Total check-ins	580
Average LOC per check-in	26
Average check-ins per developer	48

An average of 150 lines of code per file turned out to be very reasonable in practice. This is a good indicator that each action module for this system is performing one particular task, as it was intended by design.

Statistics like this can be gathered for all subsystems and plotted in a regular bar chart for comparison. At a glance, such simple graphs and statistics can help answer questions such as: How much effort is really going into extending or supporting any given subsystem? How much developer time is involved? Is more than one developer maintaining the same actions? Are development resources properly utilized? What is the cost of all this development?

5.2 Dependencies and patterns.

Workflows can also be used to detect and visualize dependencies amongst workflows based on what actions are reused and how. Figure 14 shows an extremely small sample of workflows and the names of their actions. This is remarkably close to the representation of the canonical form of a complex system [1].

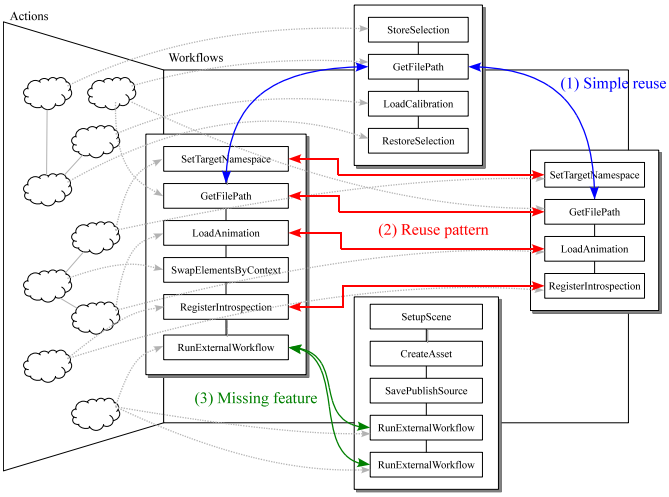


Figure 14. Module reuse, dependencies and common patterns in complex systems.

Arrows indicate actions that are reused in other workflows. They denote a *reuse dependency* between otherwise independent workflows. There are also three types of dependencies worth noting in Figure 14:

1. Simple action reuse. Provides an indication of what workflows share code without any other specific relationship.
2. Some times, there is a group of actions that appears in exactly or almost exactly the same sequence in other workflows. Cases like this can be refactored to have one workflow simply including the definition of another workflow. As opposed to duplicating the same block of actions from workflow to workflow.
3. Some actions like `RunExternalWorkflow` are normally an indication of some functionality that should be provided natively by the implementation framework. In this particular case, by declaring the exeuction of a block of actions in a detached process execution.

5.3 Code evolution

By inspecting the revision logs of each individual action module in a workflow, it is possible to visualize code evolution in the XML itself and action modules over time.

Figure 15 shows one of the most recent check-ins for a particular workflow. In the upper side, there is time slider that allows to scrub in time from the first check-in to the last. At the chosen time in the slider, each action is represented by a white rectangle that has the following information: its name; a blue progress bar providing a sense as to how quickly the action is maturing and stabilizing; and, a list of

[2] Houdini is an application used for various three-dimensional visual effects.

[3] Nuke is a high-end non-linear compositing application.

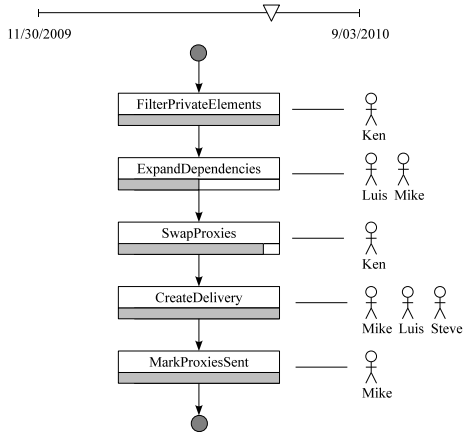


Figure 15. Code evolution time lapse. Later revision.

developers to the right of each action that have performed some modification to it in a previous check-in.

6. Conclusion

Writing code that is simple to understand, reusable and easy to maintain is challenging for individuals, let alone for multiple programmers developing collectively. Time pressure is the most common reason for cutting corners and producing code that is more cohesive than modular.

The workflow approach provides simple guidelines to facilitate writing modular code that separates data from logic to increase the chance of code reuse. This also adds structure that make programs self-documenting.

Additional benefits include the ability to monitor code reuse and patterns at any given time. Highly reusable actions can be identified, improved and promoted to higher levels of visibility in the framework. Code maturity can also be quantified as the amount of code that is promoted to higher levels. And, compared against the amount of code that is reused by branching on copy.

It is easy to keep track of the evolution of each individual module over time by relying on a source control system. The modification logs and people associated with them for every single module can be correlated with other actions in the various workflows to gain interesting insights on development, developer behavior and patterns. The possibilities of data mining are great from a managerial perspective to help better understand how the development effort and resources are begin spent in reality.

Acknowledgments

I would like to thank Brandon Ashworth, Chip Collier, Michael Irani and Mylène Pepe for their contributions to the implementation of the internal framework that enables this software development methodology in various areas of the production pipeline at Digital Domain. Craig Zerouni and Gregory Stoner for their support throughout this project. Jonathan Gerber and Steve Galle for their support in adopting these ideas in a major production. Mattias Bergbom, John Cooper, Amanda Hampton, Takashi Kuribayashi, Michael Morehouse, Antony Serenil, Blake Sloan, Geoff Wedig and many other developers who contributed invaluable feedback by using these ideas, methodology and framework to solve a wide variety of real problems in production. Doug Roble for his advice in preparing this paper. Many thanks to all of you.

References

[1] Grady Booch. Object-oriented Analysis and Design with Applications. Addison-Wesley. Second Edition.1994. pp. 14, 42.

[2] Robert Glass. Facts and Fallacies of Software Engineering, Addison Wesley, 2003, pg 120.

[3] Donald Knuth. ”Literate Programming (1984)” in Literate Programming. CSLI, 1992, pg. 99.

[4] Barbara Liskov. OOPSLA Keynote: The Power of Abstraction. Dec 23, 2009 http://www.infoq.com/presentations/liskov-power-of-abstraction.

[5] D.L. Parnas. Information Distribution Aspects of Design Methodology. IFIP Congress, 1971.

[6] Bjarne Stroustrup. The C++ Programming Language, Addison Wesley, 2000, pp. 694, 695.