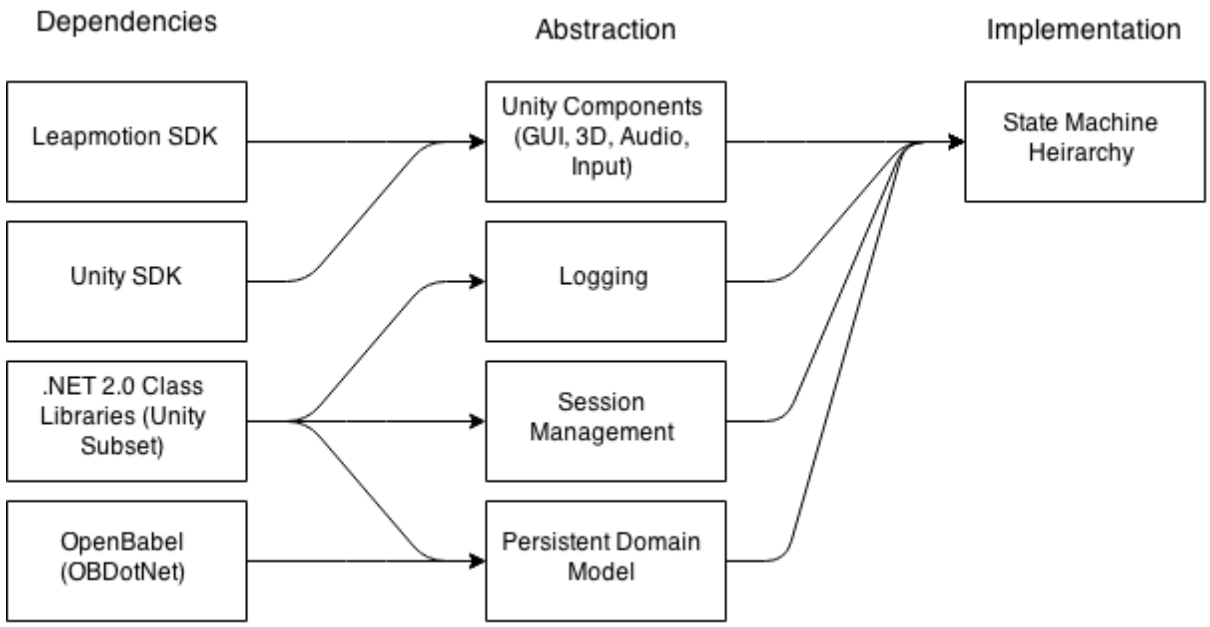
“MolyPoly 2” Software Architecture

Friday 13th of February 2015

Phillip Warren

- Implementation Layer
  - State Machine Hierarchy
    - Example abstract node class:
- Abstraction Layer
  - Unity Components
  - Logging
    - Example Logger implementation:
    - Why not use a pre-existing library for logging?
  - Session Management
  - Persistent Domain Model
    - Example High Level Domain Model Object:
    - Chemistry-Specific Domain Model
- Proposals for near-future work
- Resources

Modular Dependency Diagram



This diagram illustrates the explicit information flow between modules. Each module may only use symbols from a module that is connected by an incoming arrow.

This diagram is deliberately organised into layers from left to right. Each module may only use symbols from modules in the previous layer(s). This restriction makes it easier to reason about explicit information flow during development.

A weaker restriction would if the dependencies formed a directed acyclic graph, but this is probably too liberal for the purposes of this project, though could be used in special cases such as logging. (If for example a GameObject needs to log its position in 3D space every frame in all contexts.)

This diagram should be updated to reflect any additional modules and connections during the project lifetime.

Implementation Layer

State Machine Hierarchy

This layer consists of a FSM which controls the program at the top most level and any associated utility functions. Each “state” in this machine will appear as a node on the program FSM diagram (yet to be developed.) This diagram is intuitively hierarchical if some node (such as “show the lesson select menu”) contains within it lower level FSM. To avoid confusion with the actual state of the program, the word Node is used to mean an instance of the base class FSMNode<T> where T is the type of the parameter which is passed to the node when it is activated.

Example abstract node class:

```
abstract public class FSMNode <T>
{
	abstract protected void Enter(T context);
	abstract public FSMNode<T> Execute();
	protected FSMNode(T context) {
		Enter(T);
	}
}
```

Each node has a two functions which must be defined: Enter and Execute. Enter is called once when the node is activated (with some context provided) and Execute is called once per frame while the node is active. Execute is called once per logical frame, which does not necessarily correspond 1-1 with a rendering frame. (Code which must execute once per rendering frame belongs elsewhere).

During the process of translating the state diagram into code, it is important to explicitly state any necessary preconditions (constraints before Enter is called), invariants (constraints that are maintained at the beginning and end of every Execute call) These constraints should be stated explicitly in documentation. Implementations should only raise exceptions when the constraints are not met.

The general design of a single node is as follows:

- Enter: perform setup; i.e meet the invariant conditions in terms of the preconditions.
- Execute: do a single step of processing;
    1. (Optional, but must be documented as an assumption) Check Generalised Invariant: that the domain model is consistent with the Unity component graph.
    2. (Optional, but must be documented as an assumption) Check Specific Invariant: that there is some condition specific to this node which is met.
    3. Determine the necessary context:
        a. Read the domain model for static (persistent) state.
        b. Read the Unity Components for input or dynamic objects (such as physics objects)
    4. Update the state as necessary

    a. Update the domain model (note that the invariant is now broken)
    b. Update the Unity scene (rebuilding the invariant condition)
5. Log data
    a. Log any dynamic (per frame) state that needs to be logged.
    b. Log any transitions in the domain model.
6. Determine the next state.
    a. self: do nothing, since invariant is already met.
    b. other: meet the necessary preconditions, construct and pass in the context.
7. Return the reference to the next state object.

In this design, it is the responsibility of each state to meet the preconditions of every output state before passing control. This makes it easier to add new connections between nodes because the successor node does not need to be updated.

Abstraction Layer

Unity Components

Classes in this module depend on the Unity SDK and LeapMotion SDK.

This module is a collection of independent classes which extend MonoBehaviour. They can be composed together to represent the current domain model to the user. The job of instantiating these objects and of composing them into specific scenes is left to the implementation layer. (The unity editor can be used to create prefabricated object hierarchies which can be instantiated by an FSM node.)

Examples of classes in this layer are;

- 3D scene objects such as spheres (for representing atoms), sticks (for representing bonds), particle effects (for representing a change in the domain model such as a new connection, as an example)
- 2D GUI objects such as styled buttons, text and pictures. Specific GUI layouts can go in this module if they do not change in response to the domain model changes.
- Audio feedback
- Hand models which can change directly in response to the LeapMotion virtual hand.

Logging

Classes in this module should only depend on the .NET subset supported by Unity.

The logging module transparently (i.e, without user interaction) collects events, formats them in an ASCII format (such as CSV) pushes them on an opportunistic basis to a remote location to be collated.

Example Logger implementation:

```
using System.IO;
public class Logger {

	//Objects of this type must only access immutable data to be thread safe.
	public delegate void Loggable(StreamWriter);

	//Log without copying memory to another intermediate string.
	public void Log(Loggable s) {
		///...
	}

	//Log some string.
	public void Log(string s) {
		Log(sw => sw.WriteLine(s));
	}
}
```

Each log event is a thread-safe delegate which writes to a file when called. These references are pushed onto a queue and serialised. Because the delegate call happens in a separate thread, it is a condition that the delegate only access immutable data in order to prevent race conditions. Therefore the best way is to use special purpose or domain model objects that are immutable.

At some future time (such as at session exit) this data is compressed and uploaded to an external server.

Why not use a pre-existing library for logging?

For the purposes of this project, an industrial strength logging library such as Log4Net would be overkill and is probably a distraction. A logging library such as Log4Net is designed for logging errors on a best-effort basis with complex hierarchical filtering for debugging purposes. In our case filtering is not needed; the specific requirement is to

1. Store logs in a compressed ASCII format in a single file per session.
2. Upload all previously unuploaded logs on an opportunistic basis. The most recent log is uploaded at the end of a session.
3. Delete from disk only those logs which have been confirmed to be successfully uploaded.

Session Management

This module retrieves and restores a user session (in the form of a byte array) from disk or a remote source (via http) using an identifying code as a lookup key. Care should be taken to ensure this operation does not block the user interface from updating. Associated MonoBehaviours can be used to update the GUI during this operation. The final result can be deserialized to a Persistent Domain Model object.

Persistent Domain Model

This module consists of a set of datatypes with pure functions to represent valid transitions between program states. These datatypes are used to implement three features:

- Session to session persistence (for which serialisability is necessary)
- Undo persistence (for which immutability is necessary).
- Loggable events (for which immutability is necessary for thread safety).

Because of this immutable design it is also easy to make logging functions which use this data. (However, this module is left as side-effect free; the actual logging is done within the implementation layer since we are interested in actual user actions not domain model transitions.)

Example High Level Domain Model Object:

```
using System.Collections.Generic;
public class DomainModel {
      private SessionData session;
      private ProgramState currentState;
      private Stack<ProgramState> undoStack;

      public NextState(ProgramState s);
      public ProgramState Undo();
      public ProgramState Current();

      public SerialiseSession(BinaryWriter sw) { … };
      public DeserialiseSession(BinaryReader sw) { … };
}
```

The Undo feature is implemented as a stack of program states. When this stack is empty the undo function is disabled and simply returns the current state.

The separation between SessionData and ProgramState exists because not all state needs to persist between sessions (such as an incomplete lesson) and there is no undo feature for session data (such as completing a lesson). If it is desirable that every domain transition including the undo stack should persist between sessions, the DomainModel itself can be made serialisable.

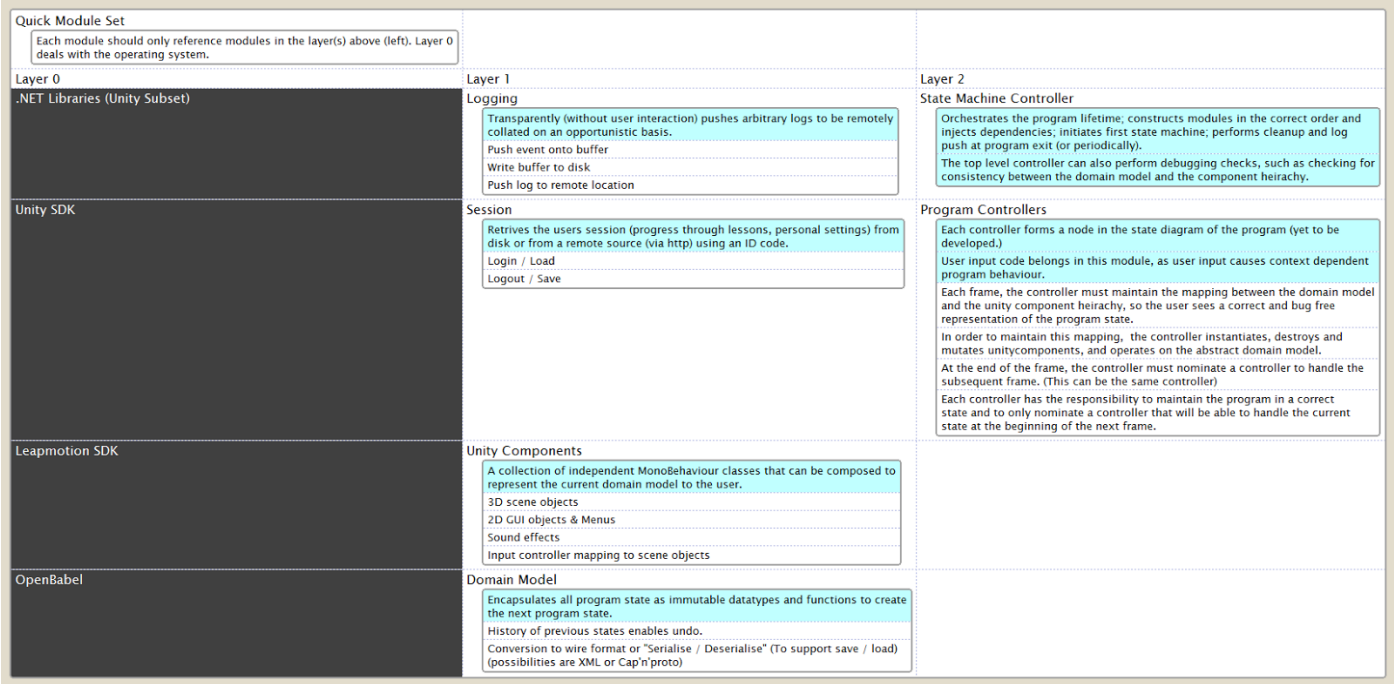
Chemistry-Specific Domain Model

The OpenBabel project implements cheminformatics tools such as parsing molecule specifications in standard formats, and writing a molecule object to a string specification of that molecule. This can be used to quickly define new molecules, to lay out a molecule in 3D, to test whether the user has constructed specific molecules, and other chemistry specific queries.

Proposals for near-future work

- Set up hosted version control.
- Create initial project & file structure.
- Develop high level state machine diagrams.
- Develop & implement domain model (some will be derived from state machine diagrams).
- Investigate user security model (to ensure ID code integrity and therefore log integrity and session integrity)
- Implement sessions
- Implement logging
- Develop UI & interaction designs
- Implement UI components
- Implement nodes in state machine diagram, document preconditions and invariants.
- Connect nodes in state machine diagram.
- Iterate on this document.

Quick Module Reference



Resources

.NET Serialisation

- File Class
- Basic Serialization
- Persistent Storage
- BinaryFormatter Class

“The serialization architecture provided with the .NET Framework correctly handles object graphs and circular references automatically. The only requirement placed on object graphs is that all objects, referenced by the serialized object, must also be marked as Serializable”

.NET Threading

- Threading Tutorial
- Queue.Synchronized Method

HTTP Requests

- Unity Simple Access to Web Pages
- WebRequest Class

Hashing

- MD5 Class

Compression

- GZipStream Class

Cheminformatics

- Open Babel: The Open Source Chemistry Toolbox
- http://openbabel.org/docs/dev/FileFormats/SMILES_format.html

Logging

- http://logging.apache.org/log4net/

Unity Mono Compatibility

- http://docs.unity3d.com/410/Documentation/ScriptReference/MonoCompatibility.html

LeapMotion C#

- Unity C# examples
- Setting up a project