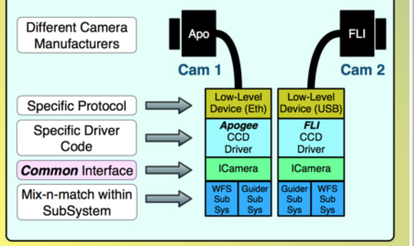
Design Principles

- Component oriented design
- Object oriented programming techniques
- Physical world is composed of components and modules that are connected with "wires"
  - Each wire represents an interface (TCP/IP, etc...)
- Common code throughout
  - Provides base classes for all objects in system
  - Similar hardware drivers implement the same generic interface class
  - Shared algorithms, routines, utilities, etc
- Scriptable interface built in at device level
- Each individual control system:
  - Controls low-level device drivers to provide I/O to the hardware
  - Provides abstraction interface to the devices so changing hardware manufacturer is simple
  - Defines it's API using Actions with a marshaled interface to the action processing system
  - Run programs that use actions to manipulate the hardware sub-systems
  - State machine driven

Use of Interface Classes to Abstract Hardware



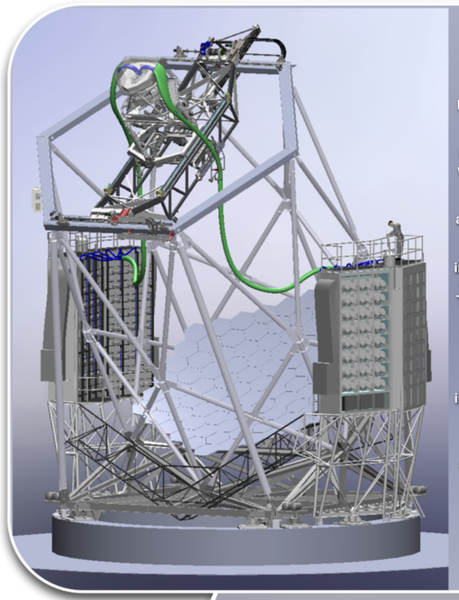
Device Managers (Factory Method Pattern)

Role of the Device Manager

- Create, setup and destroy all devices, thus encapsulating any complexities that go along with these procedures.
- Provide methods for accessing a particular device
- Abstracts the actual type of the device since the access methods return interface class types

Development Environment

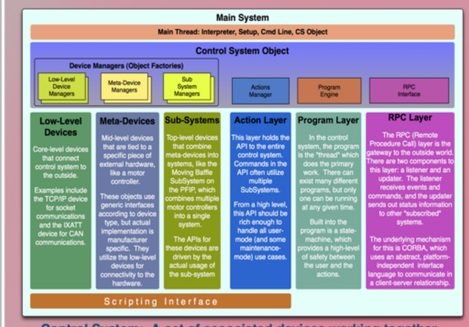
Primary Platform	RHEL 64-bit
Languages Used	C/C++, Python, tcl/tk
Toolchain	gcc, gnu make, bash, lapack, libtool, automake
Technologies	SWIG, omniORB, omniORBpy, SLALIB, pySlalib, CFITSIO, pyFits, libedit, sqlite, cJSON, log4cplus, Qt4/pyQt, Mayavi/OpenGL, Chaco, NumPy, SciPy, VTK, SetupDocs, Pyrex, Sphynx, Greenlet, GSL, MongoDB, Stemming
Continuous Integration	Nightly builds using Hudson
Version control	git with a central 'master' repository
Other	Automatic building of auxiliary libs; use of bugzilla for bug tracking; wiki used to capture specs, brainstorm amongst developers, document processes



Abstract

The Hobby-Eberly Telescope at the McDonald Observatory is undergoing a major upgrade to support the Hobby-Eberly Telescope Dark Energy Experiment (HETDEX) and to facilitate large field systematic emission-line surveys of the universe. An integral part of this upgrade will be the development of a new software control system. Designed using modern object oriented programming techniques and tools, the new software system uses a component architecture that closely models the telescope hardware and instruments, and provides a high degree of configuration, automation and scalability. This poster covers the overall architecture of the new system, plus details some of the key design patterns and technologies used. This includes the utilization of an embedded python scripting engine, the use of the factory method pattern and interfacing for easy run-time configuration, a flexible communication scheme, the design and use of a centralized logging system, and the distributed GUI architecture. The project is currently well into the implementation phase, with end-to-end systems running a combination of actual and simulated hardware. Installation and commissioning at McDonald Observatory is anticipated to begin in the fall of 2011 with science observations to start in early 2012.

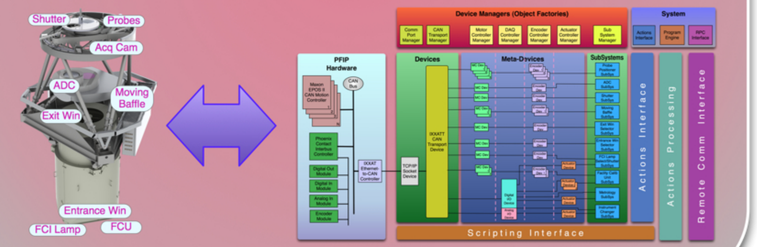
Control System Architecture



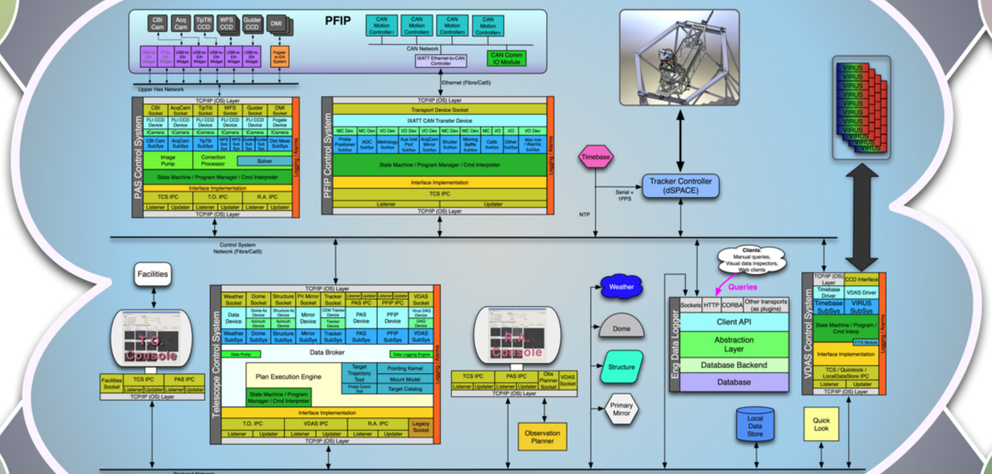
- Used as basic architecture for most systems
- Configured at run-time by scripting engine
- Scriptable down to the lowest level (hardware)
- Runs standard Linux with no special hardware
- Provides hardware abstraction using interfaces
- Phased Startup:

1. Server startup, context object, interpreter
2. CS object instantiation, managers are created
3. Pre-wiring: All managers instantiate their devices and device objects get pre-wiring setup
4. Wiring: Managers "wire" all their devices
   - All low-level devices establish comm with hardware
   - Meta-devices are configured with low-level devices
   - Sub-Systems are configured with Meta-devices
   - Post-wiring configuration of devices
5. Actions layer enabled
6. CORBA/RPC layer enabled
7. Default program is loaded and program loop starts

Control System: A set of associated devices working together



Actual Hardware... **Modeled by Software Control System**

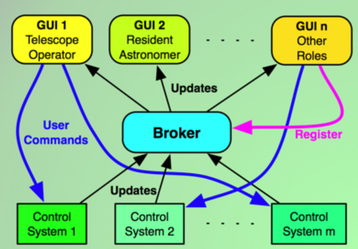


Communications / Messaging

In order to keep the communications and messaging between components and major systems as flexible as possible, we use JSON documents. JSON (JavaScript Object Notation) is a lightweight, data-interchange format that is easy for humans to read and write. By using JSON as the payload in our communications, we can easily add or modify commands without having to re-factor the interface.

```
{
  "SourceCS"  : "TCS",
  "TargetCS"  : "PFIP",
  "SubSystem" : "GuideWFS",
  "Action"    : "SetTrajectory",
  "Arguments": {
     "probe": "1",
     "trajectory": [
        {"r" : "3.1234", "theta": "2.343" },
        {"r" : "3.4323", "theta": "2.121" },
        {"r" : "3.1234", "theta": "2.343" } ]
  }
}
```

Distributed GUI

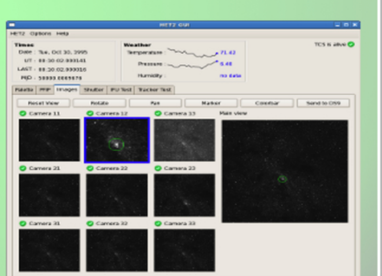


Data Flow

- Data model in control systems, not GUI.
- Each control system sees only one GUI (a broker), simplifying their operation.
- Broker distributes data to multiple GUIs as it is updated (no use for redundant model).
- GUIs send commands to control systems; control systems respond to broker; and broker distributes back to client GUIs.
- Each GUI has a common dashboard at top, but the remaining components can be customized according to the role of the user.

User Interface

- Qt/PyQt with custom widget library.
- Mayavi using OpenGL to render images and 3D visualizations in real time.
- Image visualizations allow standard set of operations in a DS9-like fashion. User can send images to DS9 for further processing, if necessary.
- User configurable set of modules (tabs) and visual elements at a given time or for a given job role.
- Fully interactive stripcharts to display data. Smaller versions (sparklines) displayed side by side with important values to quickly show trends in a glance.



Sample GUI showing Dashboard at top, and specific tabs below

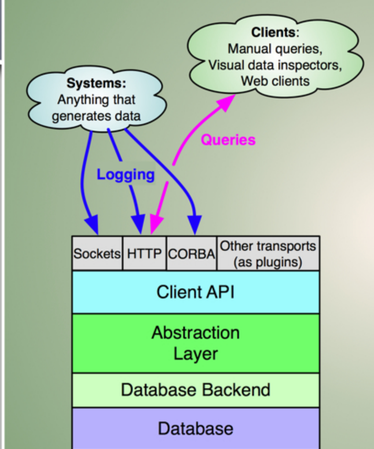
Scripting Engine

- All devices, managers, and actions available to scripting engine through the 'magic' of SWIG wrapper
- Control system is configured entirely from within scripting layer which executes shortly after startup
  - This allows the capability to completely change the configuration of your system simply by using a different or modified configuration script
  - Testing/debugging is simplified because you can easily isolate any driver or change it's properties
- Program code can start out as a prototyping script to allow agile development without changing any of the core driver code or algorithms
- Use of Python as interpreter brings in a very rich and well tested set of modules for extending the base functionality of the control system, with little or no modification to the base code

Example of python script for configuration

```
...
TrkTelescopeMgr = TelscopeCS.getTrkTelescopeMgr()     # get tele manager from main CS
TrkTelescopeMgr.setName("TrkTelescopeMgr")            # give it name for logging, etc
TrkTelescopeMgr.setNum(1, c)                          # set number of devices of type telescope
TrkTelescopeMgr.setFile(0, "/lib/tcs/drivers/TrkTelescopeHET.so", c)  # set so file name for device
...
trkTelescope0 = TrkTelescopeMgr.getObj(0, c)          # get a telescope object from tele manager
trkTelescope0.setName("trkTelescope_HET")             # give it name for logging, etc
trkTelescope0.setEastLongitude_deg( 255.985258, c)   # set parameters...
trkTelescope0.setLatitude_deg(       30.681436, c)
trkTelescope0.setElevation_deg(      55.00, c)
trkTelescope0.setHeight_m(           2003.00, c)
trkTelescope0.setFs_mm(              12526.0, c)
...
```

Central Logging



Network Server:

- 24/7 availability allowing high number of concurrent connections.
- All major system connect to it via sockets, http, etc.
- Uniform API for all the protocols.
- Separation of network server and database engine processes through a fast internal synchronized queue.
- Uses JSON format for logging and querying, and for returning set of results.
- Supports raw unformatted strings for clients with limited computing power.
- Flexible and easy querying language.

Database Backend:

- Uses schema-less design (NOSQL).
- Database engine can be changed by writing a thin layer to adapt to a defined interface.
- Currently uses MongoDB allowing high insertion speed and easy distributed replication.
- Separation of logging and querying OS processes to minimize interference in case of overloading. Each of them can run on separate machines if necessary.

Abstraction Layer:

- Handles the logic between network and database backend.
- Does stemming analysis on text to allow fuzzy queries (plurals, conjugated verbs, etc.).
- Adds information about the client and timestamps if necessary.