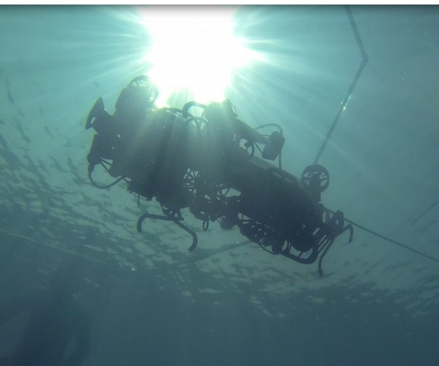
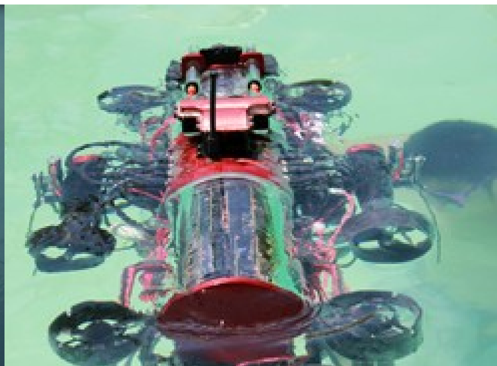
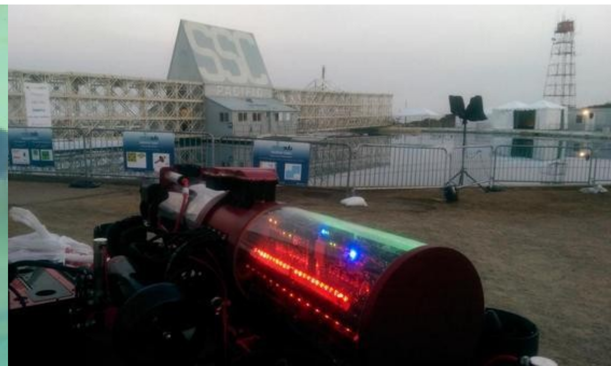
CORNELL UNIVERSITY AUTONOMOUS UNDERWATER VEHICLE

Software System Design

Christopher Goes
cwg46@cornell.edu

25 October 2014

Why?

Win the competition (duh) – but how?

Input

environment pool, tasks, Dave's ruleset
sensors cameras, acoustic – electrical team
actuators thrusters, manipulators – electrical team
vehicular platform – mechanical team

(magic) ==> make optimal decision given known information and constraints

That magic is an (ideal) software system!

What?

Programming:

Algorithms

how to maneuver around the pipe

Correctness / Bug-testing

hit the red buoy instead of the green one

Tuning / refinement

changing vision parameters

(as opposed to)

Software System Design:

The ideation and development that leads (hopefully) to an effective software structure

Segmentation

splitting a system into independent parts

Communication

enabling data transfer between separate subsystems

Writing code that will *allow* you to write code.

“Metaprogramming”

(warning: MBA speak)

“**Metaprogramming**”
maximizing the effectiveness of your programming

Base case: sub software is “a program”
The more interdependent a system is, the harder it is to test / change
(CS project difficulty ~ size squared)

Instead, split into *independent subsystems*
Developable, testable individually
Replacable with new, improved versions of similar functionality

Abstracting away *repetitive functionality*
Isolate often technically difficult / bug-ridden code
(thread-safe shared memory, sensor interfacing, etc)

(ideally, abstract any function used in multiple places)

“Metaprogramming” (part 2)

Language Choice

Choosing a programming language intelligently in the context of whatever software requirements might be pertinent

(quasi) Domain-Specific Languages

Creating “custom” syntax and abstractions to effectively describe the structure and logic of your programs

Adaptability vs Efficiency

Assembly vs Ruby, and the more-likely in-between

Coding Practices

“management” (ugh)

(but very effective if done properly)

How: Part 1 (Segmentation)

No function should do multiple things.
No program should do multiple sorts/types of things.

(generalization)

Identify distinct parts of your system

- Mission
- Vision
- Acoustic (hydrophones)
- Filtering
- Electrical interfacing
- Self-monitoring
- (etcetera)

Case Study: CUAUV

Mission System

High-level control code: what to do, when/how to do it

Vision System

Recognizes objects in forward/downward cameras

Modular – can switch “modules” (recognizers for different elements) on/off

Unified Serial Daemon

Abstracts away electrical (RS-232) interfacing to sensors/thrusters

Kalman Filter

Transforms (filters) sensor data

Controller

Transforms locational desires (speed/orientation) into thruster values

Case Study: CUAUV

Mission, Vision, USD, Controller, Kalman all (mostly) seperate pieces

Individually constructable, debuggable, updatable

“Interface” of sorts, albeit often informal
- Often just reads/writes to shared state

SHM (shared memory)
- POSIX (fancy standard) – compliant block of RAM
- Readable/writeable by multiple concurrently executing processes
  - (with proper locks, of course)
- Allows destination/source-agnostic data transfer
  - The Kalman filter doesn't need to “send” filtered data anywhere – just put it in SHM, where anything that needs to can read it.

How: Part 2 (Language Choice)

The General-Purpose “Language Hierarchy”

- Assembly
- C
- C++ / Java
- Python
- (Haskell?!)

“Robotics-directed” Custom Languages

- (remember LEGO Mindstorms)
- ROS

Tradeoff: Flexibility vs (initial) ease-of-use

Better solution: **Flexible language, custom abstraction layers**

Case Study: CUAUV

Two main languages:
- C/C++
  - Vision (although could be in Python w/OpenCV bindings)
  - Unified Serial Daemon
- Python
  - Mission
  - Kalman filter (w/ numpy)
  - Controller (w/ numpy)

Speed vs. ease of construction, modification, abstraction.

Bindings (e.g. numpy) into lower-level code for the “computational” sections, high-level languages that abstract away timesinks (memory management, complicated inheritance, etcetera) so programmers can focus on code logic as opposed to syntax or low-level implementational particulars.

(quasi) Domain-Specific Languages

Want: Flexibility of a scripting language with the coding efficiency of ROS

Thought: What is ROS? Just an abstraction layer over another language.

Instead: Define custom ("domain-specific") abstraction layer
- Optimized for your submarine
- Low-level control (when necessary) but high-level usage
- Enables full-stack control and optimization
  - (but use libraries when appropriate!)

How: Many ways
- Custom "datatypes" (functional languages, sorta Python)
- ~
- Class hierarchies (Python, C++, Java, etcetera)

(classes really are datatypes)
(Python is – unintentionally – a great example)

Case Study: CUAUV

Mission system uses custom “datatype” (through Python hackery): “Task”

Missions consist of tasks (a function from world state to sub action)

Tasks can be defined, combined, etcetera in various ways

“forward = ForwardTask()
left = TurnTask()
forward_and_left = Concurrent(forward, left)”

(or more concisely)
“forward_and_left = Concurrent(ForwardTask(), TurnTask())”

(pseudocode)

Definitions are atomic: new tasks are defined in terms of existing ones

Enables intuitively clear execution flow

Can easily change fundamental behavior
(if we wanted to go backwards instead of forwards...)

Coding Practices

(but management was supposed to be for *after* college!)

Code Reviews

Timesink
But useful to avoid (especially trivial / logical) errors
Use for critical pieces / sections of code

Version Control

A must. Git is the standard here (and for a reason).
Other options:
- SVN (inferior diff control)
- DARCS (if you feel adventurous...)

Enables easy tracking, rollback, backups, etc.
All software members have a local copy, master on some server.

(we host a custom Git repository, but Github/Bitbucket etc. also fine)

Summary

Good software system design enables good programming.

- Programmers only have to write a particular piece of functionality once.
- System-critical components are isolated and easily debuggable.
- Logic (“declaration”) is seperated from implementation (“procedure”).
- Updates can be made easily with little fear of cross-system effects.
- All software members can access systems but focus on particulars.
- Primary languages are chosen appropriately for the task in question.
- Custom “sub-languages” are used in systems wherever useful.

Coding, fundamentally, is simply abstraction.

It's all just a stream of zeroes and ones.

Abstraction has many levels ± a programming language is just one.

Done improperly, a software system can hinder functionality.

Done well, it allows the programmer to focus entirely on their specific goal and abstracts away everything else.

Questions?