OO Design Methodology
Gather Initial Documentation

- Full written technical description of the existing hardware:
  - Geometry, mass and inertia matrix
  - Power source(s), characteristics, storage, power flow
  - Things that can be controlled, and control mechanisms
  - Things that can be monitored, and monitoring mechanisms
  - Startup, operating and shutdown sequences, pre- & post-conditions
  - Performance limitations, constraints, restrictions and de-rating
  - Electrical/hydraulic etc. circuit diagrams
  - Maintenance manual, operating manual
  - Known faults and failure mechanisms
  - Talk to operators(s) – how to operate, what goes wrong...

- Output: Hardware Description document as a concise shared reference of all this information
Design and Implementation Model

- Use UML for systems modelling
- Use "iterative development" (spiral development model)
  - The repeated cycle of Analysis -> Design -> Code -> Test steps produces a series of iterative prototypes of increasing capability
- Define the minimal system for a first build
- Once the first build is achieved, the system always "works"
- Successive builds add functionality
- Early risk reduction, high priority features build first, early availability, etc...
ROPES

- ROPES† = Rapid Object-Oriented Process for Embedded Systems

- Diagrams in these slides show the ROPES process and artefacts

†Douglass, B.P. Doing Hard Time. Addison Wesley, 1999
ROPES Process Artefacts

Analysis

Analysis Defects

Tested Application

Object Model

Scenarios

Use Cases

Test Vectors

Design

Design Defects

Translation Defects

Design Object Model

Application Components

Translation

Third Party Components

Real Time Framework

Legacy Code

Application Requirements

Hazard Analysis
Analysis

- Answers the questions
  - What does the client *really* require?
    (i.e. what are we going to build)
  - Can it actually be built?

- Three analysis steps
  - Requirements
  - System
  - Object
Requirements Analysis

- Begin from the client statement of application requirements, including:
  - Mission scenarios
  - Objectives and constraints
  - Assumptions, initiation and termination
  - Operational conditions - terrain, temperature, etc
  - Design objectives - rational basis for tradeoffs (speed vs. endurance, etc)
- Need to fully define a functional description of what the system must do, for client approval.
- Will often be quite different from what the client thought that they wanted…
Requirements Analysis (2)

- Define use cases
  - Identify Actors and Use Cases
    - Actor is a human or computer outside the scope of system being considered - things that can use the system
    - Use Case is a particular functionality of the system that results in return of information to an actor - how the system can be used
  - Expand to add generalisations, including «includes» and «extends»
    - Output: one or more use case diagrams
Requirements Analysis (3)

- Define events and messages
  - Nature of event/message
  - Sender, receiver, content and timing of event/messages
  - May include data streams, depending on what is the "system"
  - Output: external event list

- Define use case scenarios
  *each Scenario is one path through a Use Case*
  - Output: written description of a number of scenarios for each use case
  - Output: sequence diagram for each scenario, showing content and timing of interactions between actors
Requirements Analysis (4)

- Define behaviour within reactive use cases
  - Use statecharts to show reactive behaviour of event-driven use cases
  - All scenarios will be visible as paths through a statechart
  - Outputs: one or more statecharts

- Identify Safety Hazards
  - Uncontrolled release of energy
  - Output: list of all hazards that must be controlled in design, including severity, risk and probability of occurrence
Requirements Analysis (5)

- Write Test Set
  - Output: written specifications of a set of tests designed to verify that requirements are met.

- Write Requirements Specification
  - Analyst's understanding of the fully-specified requirements.
  - Output: requirements specification document
  - Note - may be for the whole system or for a sub-system
Detailed exploration of possible solutions

Objectives

- Find a system design that is demonstratively capable of satisfying the Requirements Specification, without having to design and build the whole system!
- Reduce risk by achieving proof of correctness and completeness at an early stage in the cycle

Solution - use executable analysis models whenever possible
Executable models can include
- Finite state machines, continuous controllers, estimators, sensor noise models, etc.
- Use *Matlab, Simulink, Stateflow*, etc. for fidelity
- Construct simulation interfaces to map onto eventual system interfaces
  - testing without hardware
  - testing with “enhanced” or “degraded” hardware
- Generate final algorithms from executable analysis models
  - minimal discard of code
  - "the model is the system"
Systems Analysis (3)

Start with a preliminary cut into likely functional subsystems

- Sub-system is a large organisational unit containing (usually) hardware and software - e.g. navigator

- For each subsystem, define the interfaces to/from each hardware and software component
  - high-level definitions, based on information flow, form of information, data rate, etc.

- For each sub-system, make a preliminary partition between hardware and software.

- For each subsystem, define the interfaces to/from each hardware and software component
  - again, high-level interface definitions only
System has now been tentatively partitioned into a number of subsystems, with each subsystem containing one or more software or hardware (electronic and/or mechanical) components.

Conduct detailed analysis on each high-risk subsystem (some may be simple and straightforward)
- analysis method depends on subsystem under analysis...
- base assumed hardware performance levels on experience, consult with potential vendors
- estimate code size and speed (very hazardous!)
Systems Analysis (5)

- Continue subsystem analyses until it is clear that this set of partitions will satisfy the requirements

- Otherwise, do new subsystem and/or software/hardware and/or mechanical/electronic partition, and repeat...
Outputs from Systems Analysis

- **Output:** Deployment diagram showing the computer boxes with their software components, interfaces and sensor/actuator hardware
- **Output:** Hardware Specification - detailed statement of responsibility and performance specifications of each hardware component
- **Output:** Software Specification - detailed statement of responsibility and performance specifications of each software component
- **Output:** Mathematical models used in the analysis of estimators, control systems, etc
- **Output:** Simulation results, tuning results
- **Output:** Executable state machines (we don't do this presently, but should...)
- **Output:** Test set - written specifications of a set of tests designed to verify that requirements are met.
- **Output:** Test plan

Note that the solution has not yet been determined, although we should be happy that the system can be built.
Object Analysis

So far, functional and behavioural views of the system have been defined.

- Next, all of the objects and classes that are essential to implementing a solution are identified.

- For large systems, split the analysis into a number of Domains to allow work in parallel.

  A Domain contains a group of related classes that collaborate to achieve part of a solution.
Structural object (static) analysis
- Identify the essential objects
- Identify the classes, and collaborations between the classes
- **Output:** class diagrams of static classes
- Refine class diagrams to show collaboration, increasing detail

Behavioural object analysis
- Define the essential behaviour of reactive objects
- Identify the class to which each statechart belongs
- **Output:** class diagrams of classes containing statecharts
- One useful test: "walk through" the use case scenarios using the identified objects.
Design

- Given the understanding that comes from analysis, we need to specify (choose) one solution that is consistent with the analysis.

- Design is the process of identifying one solution that "best" meets all of the requirements – no new "surprises"

- Three stages of design
  - Architectural: system-wide - processors, packages, components, tasks
  - Mechanistic: groups of collaborating classes
  - Detail: design at the individual class level
ROPES Design Model
The "big", strategic decisions that affect all of the system:

- Distribution of software components across processors & processes
- Identification and definition of processes and threads
- Choice of concurrently and scheduling models (in association with choice of OS)
- Implementation patterns for active objects (root threads) and state-based objects
- Inter-processor comms, including protocols and hardware layers
- Implementation pattern for error handling
- Implementation pattern for safety system (homogeneous redundancy, diverse redundancy, actuator-monitor, etc)
- Fail-safe actuator design
- Fault detection and recovery policies
Outputs are refined and extended versions of the analysis diagrams:

- Output: **class and object diagrams**, including depiction of architectural patterns
- Output: **collaboration and/or sequence diagrams**
- Output: **statecharts**
Mechanistic design adds standard design patterns (container-iterator, smart pointer, etc) to connect the main architectural elements together.
Implementation (Translation)

- Translate the various models into source code, compile and unit test.

- Can be manual
  - Use frameworks and “style guide”

- or automatic (e.g. Rational Rose, Ilogix Rhapsody)
  - Expensive, learning curve
  - Model stays in synch with the code…
“Unit testing” is thorough testing of a unit of executable code before it is added to the system.

- A unit can be as small as one class, or as large as a group of collaborating classes that form an executable subsystem.

- A unit can include sensor, actuator or other hardware.

- Objective of unit testing is to ensure that the unit implementation is consistent with the design.

- Unit testing is “white-box” – have to understand (and often delve into) what is inside in order to test it.
Testing

- Testing proper begins when unit testing ends

- Integration testing
  - “Gray box” – some understanding of internals required

- Validation testing
  - “Black box” – testing at the system boundary
ROPES Testing Model
Testing (2)

Integration Testing
- Add a component to the system, and test interfaces (new public functionality added)
- Conducted by build master or component developer

Validation Testing
- Test against validation tests derived from previous requirements and analysis documents
- Ideally, a separate test team designs and runs validation tests
- Safety system validation: go “inside” the system and systematically break one thing at a time.
Test phase results in

- Output: **Validated Component**: software/hardware component increases system functionality
- Output: items added to **Defect List**

- Defects are folded back for a further analysis – design – implement – test cycle…