Architecture III
Uses and Process Structures

Outline

- Lecture:
  - Design for extension and contraction: the “uses” structure
  - Design for run-time separation of concerns: the process structure
- Lab Exercise: wave-height implementation

Purposes of System Subsets

- Often good reasons for extending or contracting software capabilities
- Extensions
  - Planned upgrades in new versions
  - Develop system as a set of increments, each adding capability
- Contractions
  - Build to schedule, e.g., time-to-market means it’s better to deliver reduced capability on time than full capability late
  - Provide lower cost, lower capability subset (e.g., a “basic” version)
  - Repurpose a subset of system for a related development
Difficulties

- Where subsets and extensions are not planned for, likely difficult to do
  - Removing capabilities results in other components not working
  - Capabilities cannot be added without changing existing system modules (e.g., adding/changing services)
  - Extending or contracting requires redesign
- Problems follow from unplanned dependencies
  - Arise by default during development (e.g., when creating functional behavior for use cases)
  - Module developers are free to use the services of every other module
  - Little thought given to downstream implications

Uncontrolled Dependencies

- Result of unplanned development is typically a network of dependencies (undirected graph)
- When will I have a working system?
- What happens if I need to deliver a subset?
- What do I want this to look like?

The "Uses" Relation

- Relationship is formalized as the "uses" relation
- Definition: Program A uses program B if a correct version of B must be present and working correctly for A to work correctly
  - Intuitively: Any system with A in it must also have B if A is to work correctly
- "uses" is defined over programs (e.g., services) but may be simplified as a relation between modules
- Often the same as "calls" but not always
  - A may call B but not use it (would work with a stub)
  - A may use B but not call it (B produces data or performs services A needs, garbage collection)
As Architectural View

- The “uses” structure exists whether any thought is put into it or not
- The structure affects a range of important system and development qualities (hence design goals)
  - Ability to deliver increments
  - Ability to extend/contract capabilities to meet schedule
  - Portability (layers, abstract machines)
  - Testability (incremental build/test)
- Meeting these kinds of design goals requires purposeful design of the “uses” structure
- The “uses” as architectural structure
  - Components: services or modules (depending on granularity)
  - Relation: “allowed to use”
- Specifically, the assumptions that A makes about B

Uses Hierarchy

- “Ideal” design gives “loop-free” hierarchy with uses relation (acyclic tree)
  - Level 0 uses nothing else
  - Level N only allowed to use services on N-1 (or below)
- Defines constraints on
  - Build/test order
  - Increments & subsets
  - Layers
- Other design concerns may result in difference from the ideal

FWS Uses Relation On Modules

Controller

Message Generation

Value Generator

Transmitter Device Driver

Message Format

Data Banker

Sensor Monitor

Sensor Device Driver
Tabular Representation

<table>
<thead>
<tr>
<th>Uses</th>
<th>Map oversees</th>
<th>Map oversees</th>
<th>Tile.makes</th>
<th>Tile.makes</th>
<th>CM.shrink</th>
<th>CM.RegionIterator</th>
<th>CM.tile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fit in Development Process/Plan

- **Requirements**
  - Defines expected evolution (versions of the system)
  - Defines required subsets
- **Project planning**
  - May identify additional requirements or constraints
  - Specifies increments (trace to requirements)
  - Specifies build/test order
  - Project plan should map subsets/increments to schedule
- **Architectural design and specification**
  - Decompose the system into modules, services
  - Design uses relation to support subsets & increments
  - Represent "uses" design decisions (e.g., in table)
  - Individual modules document what they use from other modules

DSD Concerns

- **Dependencies between components/tasks**
  - Uses implies a dependency between components that may or may not be obvious
  - May require communication, imply changes
  - Should be made explicit
- **Effect on incremental development**
  - Need to think ahead about capabilities each increment will need
  - Necessary for project planning
Importance of Uses (1)

- Uses determines the order in which modules should be implemented
  - Data Banker & Sensor Device Driver Before Sensor Monitor & Value Generator

Remember that Uses and Module Structures Are Different

Importance of Uses (2)

- Uses determines the modules that are needed to build a family member
  - Especially for building subsets or increments of the system uses tells you what must be present and which should be built first
    - If message generation is included, then so must be Value Generator,
      Transmitter Device Driver, Message Format, Data Banker, Sensor Device Driver

Uses Design Heuristics

- General considerations to decide if A should be allowed to use B (above B in uses-hierarchy)
  - A can be made simpler if it uses B (and B would not be simpler using A)
  - There is a subset/increment that needs capabilities provided B but not A
  - B is part of a coherent virtual machine layer that A uses
  - Using B allows A to keep its secrets (e.g., modules A & C don’t need to share information about a data structure if they both use B)
Summary

- Must design a system to address increments, subsets, layering
- Difficult when dependencies are unplanned
- Represented in the architecture as the "uses" relation
  - Makes dependencies explicit
  - Can design and plan for increments, etc.
  - Should be traceable to project plan

Process Structure

Process Structure(s)

- Process: A sequence of events
- Processes may cooperate in a variety of ways
  - Sending messages
  - Synchronizing
    - Sharing resources
    - Producing and consuming (data)
    - Event notification
      - Publish and subscribe
      - Observer pattern
  - Different modes of cooperation may lead to different relations and therefore different structures
Process Structure Goals

- Timing requirements (performance): periodic and demand on outputs
- Portability: multi-processor platform
- Feasibility, understandability, maintainability
  - Ability to treat different run-time threads as relatively distinct
  - Easier to write, understand
- For DSD
  - Architectural view of run-time dependencies
  - Design to reduce run-time dependencies between code developed by different teams

Components & Relations

- Components are processes (aka tasks, threads)
- Relations of interest:
  - Synchronizes-with: process A must synchronize execution with B
  - Excludes: process A shares an exclusive resource with B
- The scheduler uses these relations to generate a schedule

FWS Process Structure Representation

- Message Generator
- Sensor Monitor
- Data Banker
- Read
- Write
FWS Sequence Diagram

Message Generator → Data Banker → Sensor Monitor

Message Generator

Data Banker

Read

Write

© S. Faulk 2010

Process Structure Design

- General decomposition rule: if two tasks can run independently and concurrently, they should be implemented as distinct processes
- Time budgeting
  - Allocate timing budget to each function (demand/periodic)
  - May include
    - Demand: Initiating event, deadline, latency, execution time
    - Periodic: period, frequency, latency, deadline, execution time
- Scheduling
  - Develop feasible (or optimal) schedule for total set of processes s.t. every process satisfies its timing constraints

© S. Faulk 2010

Intersection with Module Structure

- Threads run through module services
  - Locus of control in some module
  - Calls programs in other modules
  - Together must satisfy timing constraints, etc.
- Simplify intersection using consistent design pattern

© S. Faulk 2010
Evaluation

- Ability to produce feasible schedule
- Accurate outputs produced on time
- Processes can be written or reviewed (relatively) independently

Summary

- The “uses” structure is critical to managing inter-module dependencies
  - Use to control ordering of incremental development
  - Use to understand how different teams or developers work depend on one another
- The process structure is critical to managing run time dependencies
  - Provides a way to think about and control timing dependencies
  - Separate threading supports concurrent development, testing

Exercise:
Wave Height Implementation
Implementation

- Your teams should collaborate to implement the wave height reporting feature for the FWS
- Need to agree on which teams will implement which modules
- Over the next couple of days, implement, integrate, and test
- Suggest trying something simple first like just passing data

End