Software Architecture for DSD
Part II

DSD Team

Working Definition

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.”

From Software Architecture in Practice, Bass, Clements, Kazman
Effects of Architectural Decisions (What?)

- What kinds of system and development properties are affected by the system structure(s)?
- System run-time properties
  - Performance, Security, Availability, Usability
- System static properties
  - Modifiability, Portability, Reusability, Testability
- Production properties? (effects on project)
  - Work Breakdown Structure, Project cost, time to market
- Business/Organizational properties?
  - Lifespan, Versioning, Interoperability, Target market

Which structures should we use?

- Choice of structure depends on which qualities we want to design for
- Compare to architectural blueprints
  - Different blueprint for load-bearing structures, electrical, mechanical, plumbing
  - Designing for particular qualities requires the right view
    - Process structure for run-time property like performance
    - Module structure for development property like maintainability
- Implication: Schematic Map structures should address major project risks
  - Communication overhead and uncertainty
  - Fixed deadlines
DSD Architectural Design Goals

- Limit the necessity for communication: decompose the system into parts such that
  - Each part can be assigned to a different team and developed independently
  - Parts can be independently verified
  - Properties of the system that are likely to change (e.g. implementation details) are encapsulated
  - Only properties of the system that are unlikely to need to be change are part of the architecture
  - Role of each part in the overall system is clear (and when together, implement the requirements)

- Support incremental development
  - Can identify a subset of parts that implement a useful subset of the system
  - Can add capability by extending the working system

Key Architectural Structures

- Module Structure
  - Decomposition of the system into work assignments

- Module Interface Specs
  - Define Services Provided and Services Needed
  - Define syntax and semantics for accessing services
  - Define data types, program effects, ...
  - Define test cases
  - Record design decisions and implementation notes

- Uses Structure
  - Describes the allowed “uses” relationships between modules and limits what other modules the implementer of a module may use.
Modularization a la Parnas

- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a “module.”
- A module is characterized by two things:
  - Its interface: what the module provides to other parts of the systems. Properties other parts of the system may assume and services they can use.
  - Its secrets: what the module hides (encapsulates). Properties other parts of the system may not use and should not depend on.
- Modules are abstract, design-time entities that may or may not directly correspond to programming components like classes/objects.

Role of Modules

- Modularization affects broadest set of system qualities
- Communicate/verify
  - Can understand system one component at a time
  - Can review or test individual units before the whole
- Managerial
  - Provides units for work-breakdown, scheduling
  - Allows concurrent development
- Coding
  - Can write modules with little knowledge of other modules
  - Replace modules without reassembling the whole system
- Flexibility/Maintainability/Reuse
  - Anticipated changes affect only a small number of modules (usually one)
  - Can calculate the impact and cost of change
  - Provides unit of reuse
Did quick review against design goals

Basic questions:
- What structures are represented in this view
- How do we create structures that satisfy our design goals?
- How do we know if we have succeeded (evaluate the design)?

Modular Structure

- Comprises components, relations, and interfaces
- Components
  - Called modules
  - Leaf modules are work assignments (usually)
  - Non-leaf modules are the union of their submodules
- Relations (connectors)
  - submodule-of => implements-secrets-of (also “uses”)
  - The union of all submodules of a non-terminal module must implement all of the parent module’s secrets
  - Constrained to be acyclic tree (hierarchy)
- Interfaces (externally visible component behavior)
  - Defined in terms of access procedures (services or method)
  - Provide only external access to internal state
Module Structure Design Goals

- I.e., desired properties of the module decomposition, interfaces, and relations
- Limit the necessity for communication: decompose the system into parts such that
  - Each part can be assigned to a different team and developed independently
  - Parts can be independently verified
  - Properties of the system that are likely to change (e.g. implementation details) are encapsulated
  - Only properties of the system that are unlikely to need to be change are part of the architecture
  - Role of each part in the overall system is clear (and when together, implement the requirements)
General Approach

- General approach to module decomposition based on principle of “information hiding”
- Information hiding: design principle of limiting dependencies between components by hiding (encapsulating) information other components should not depend on
- An information hiding decomposition is one following the design principles that (Parnas):
  - System details that are likely to change independently are encapsulated in different modules.
  - The interface of a module reveals only those aspects considered unlikely to change.

Applied IH

- IH plays two distinct roles in modular decomposition
- Guide decomposition
  - Criterion: System details likely to change independently should be encapsulated in different modules
  - Used to determine when a module should be further decomposed
  - Product: the module structure as documented in the module guide
- Guide module design
  - Criterion: The module interface reveals only those aspects considered unlikely to change
  - Used to determine what the module interface should hide
  - Product: the module interface specifications
Applying IH to Module Decomposition

- Notionally proceeds top down
  1. Choose a module and examine its secrets (module encapsulates everything other modules should not depend on)
  2. Identify aspects of the module that are likely to change independently
     1. If none, stop and define the interface (to encapsulate what can change)
     2. Else, allocate the aspects of the module that are likely to change independently to different module and repeat

General stopping criteria
- Each module contains only structures likely to change together
- Each module is simple enough to be understood fully
- Each module is small enough that it makes sense to throw it away rather than re-do

Module Structure

Would a decomposition using IH meet these goals?
- Each part can be assigned to a different team and developed independently
- Properties of the system that are likely to change (e.g. implementation details) are encapsulated
- Only properties of the system that are unlikely to need to be change are part of the architecture
IH Guide Interface Design

- The module interface reveals only those aspects considered unlikely to change
  - The only assumptions that should appear on the interfaces between modules are those that are considered unlikely to change
  - Each data structure is used in only one module
  - Data structures may be accessed by programs within the module but not by those outside the module
  - Any other program must access internal data by calling access programs on the interface

- Why do we say "assumptions" here?

Simple Example

- A simple module for maintaining account balance
- What is hidden and what isn't?
- What assumptions are on the interface?
- Is this an abstraction?
General Evaluation Criteria

- Completeness
  - Is every aspect of the system the responsibility of one module?
  - Do the submodules of each module partition its secrets?

- Ease change
  - Is each likely change hidden by some module?
  - Are only aspects of the system that are very unlikely to change embedded in the module structure?
  - For each leaf module, are the module’s secrets revealed by its access programs?

- Usability
  - For any given change, can the appropriate module be found using the module guide

Design for Extension and Contraction

The “Uses” Relation
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

© S. Faulk 2009
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”
  - Interfaces: where A uses B, what B provides that A uses
    - 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
Top-Level Schematic Map

- "Uses" between Modules

```
A map is composed of an ensemble set of layers. Each layer may be a vector layer or a raster layer.

The relationship between planar coordinate systems with uneven transformation is separate from actual manipulation of graphics.

A tile is a raster representation produced by the "render" operator on a map, with an interpretation of its coordinates.

```

- "Uses" between services

© S. Faulk 2009
Tabular Representation

<table>
<thead>
<tr>
<th>Uses Used</th>
<th>Map, render</th>
<th>Map.shrink</th>
<th>Tile.draw</th>
<th>Tile.mapCoord</th>
<th>CM.shrink</th>
<th>CM.RegionIterator</th>
<th>CM.tile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map, render</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map.shrink</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tile.draw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tile.mapCoord</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.shrink</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.RegionIterator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.tile</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</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
Particular Project Concerns

- Incremental development
  - Need to think ahead about capabilities each increment will need
  - Necessary for project planning
- Distributed Development
  - Uses implies a dependency between components that may or may not be obvious
  - May require communication, imply changes
  - Should be made explicit

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

“Uses” relation: schematic map example
“Uses” depends on what “correct” means

- A “correct” version of display depends on a correct version of tile
  - and “correct” is a consistency relation between required and actual behavior
- If we subset required functionality, we can define stub systems for development
  - Like defining program subsets in a family, but intended to check progress rather than satisfy users
Breaking a Dependence
(for a development milestone)

A correct user interface depends on a correctly functioning map tile module, which depends on other modules.

A development milestone can stub the Tile module (presenting a fixed image, with fixed coordinate mapping) if requirements of user interface and display are cut in a corresponding way.

Developing a Subsystem

The model part with parts of “Tile” can be developed and tested before integration with the GUI.

Coordinate mapping can be tested independently of everything else ... if the spec is precise and complete.
Map and Coord Mapping

Why does “Map” use “Coord Mapping”? Why not the other way?

How would the interfaces be different if “Coord Mapping” used “Map” instead?

How would you individually test Map and Coord mapping, with this uses relation and with the reverse uses relation?

GUI and Map

Can this be right? User interface and display use Tile but do not use Map, and Tile does not use Map. How does the use interface get a displayed tile?

Suppose it is accurate. Where does “View/Controller” sub-team coordinate with “Model” sub-team? How can “View/Controller” team avoid waiting for “Model” sub-team?
Review Questions For Abstract Interfaces:  

*Requirements Validity*

- For each service provided by the module, is the service valid for all expected uses of this module? If not, give an example of a use where the service is not valid.

- For each service provided by the module, is the service valid for all expected configurations and versions of this module? If not, give an example of a needed configuration or version where the service is not valid.

- For each service needed described in this specification, is a module (or set of modules) identified that this module is allowed to use to satisfy the need?
Review Questions For Abstract Interfaces:

**Requirements Sufficiency**

- Does the set of services provided specify all of the services that will be needed by users of this module? Are there any services defined that are not identified in the requirements?

- Does the set of services needed specify all of the services that this module will need from other modules in order to operate correctly? What services are needed that are not identified in the requirements?

Review Questions For Abstract Interfaces:

**Consistency Between Services Provided and Access Programs**

- For each Services Provided described in this specification, which access program(s) can be used to satisfy the service?

- For each access program and signal specified in sections 2 and 6 which Service Provided is satisfied by the access programs?
Review Questions For Abstract Interfaces:  

*Access Program Adequacy*

- Is the set of access programs and signals sufficient to satisfy the uses needs of modules that are allowed to use this module?

- Are there access programs that should be combined into one access program?

- Are there single access programs that should be refactored into several different access programs?

- Are the performance requirements adequate for the uses that will be made of this module?

---

Review Questions For Abstract Interfaces:  

*Implementation of Variability*

- Which variability (or variabilities) does this module implement?

- Are all values in the range defined in the parameters of variation accounted for in implementing each variability? Which values are not?

- Can the variabilities be bound at the time specified in the commonality analysis for the variabilities?
End of Exercises