Software Architecture for DSD:
The “Uses” Relation

- Design for extension and contraction
- The “uses” structure
- Role in incremental development

Architecture and DSD

- Problems of coordination and control are affected by the way the code is structured
  - Problem of distributing work: want to be able to have different sites develop code concurrently without increasing communication overhead
  - Problem of incremental development: need to coordinate development so all the pieces are developed in the right order for each increment
  - Problem of run-time dependencies: timing dependencies at run time also require communication
Meaning of “Architecture”

“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.” - Bass, Clements, Kazman

- Systems have more than one architecture
  - There is more than one useful decomposition into components and relationships
  - Each addresses different system properties or design goals

Choose Structures

- Module Structure
  - Decomposition of the system into work assignments or information hiding modules
  - Most influential design time structure
    - Modifiability, work assignments, maintainability, reusability, understandability, etc.
- Uses Structure
  - Determine which modules may use one another’s services
  - Determines subsetability, ease of integration (e.g. for increments)
- Deployment structure
  - Determine constraints on how the system will be deployed
  - Constraints on platforms, networks, etc.
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
Uncontrolled Dependencies

- Unplanned subsets and extensions likely difficult to do
  - Removing capabilities results in other components not working
  - Capabilities cannot be added without changing existing system modules
  - 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

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 (meet its requirements)
  - 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, e.g. 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
  - 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, whatever B provides that A requires
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

Uses Hierarchy

- For incremental subsets
  - First subset using nothing else
  - Second only uses first or itself
  - Etc.
- Planning: if each is a module, then know the set of work assignments for each subset
- Key point: the “allowed to use” relation is something the architect designs (not just a view of an existing system)
Face Recognition Uses
Increment 0

Which programs “use” others?

Client App

\[ \text{recognize (img)} \]

\[ \text{train} \]

\[ \text{recognize} \]

\[ \text{calls} \]

\[ \text{uses} \]

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
    - 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
  - Use 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 by 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

Questions?
When do we “use” the DB?