Software Architecture for DSD

- The problem of distributing work
- Review: the role of architecture in determining system qualities
- Module structure: designing components and interfaces for distributed teams

Code Structure and DSD

- Problems of coordination and control are affected by the way the code is structured (decomposed into parts and the relationships between the parts)
  - 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 require synchronization
- Focus on systematic approaches to architectural design
Problem of Distributing Work

- What kinds of issues arise when different sites develop different parts of the system?
  - Need to divide the system into parts
  - Have parts worked on concurrently by distributed teams
  - Have a working system when the parts are put together for testing or integration (each increment)

Implications of Code Structure

- In practice, there are always dependencies between system parts
  - Data dependencies (component C1 uses data produced by C2)
  - Functional dependencies (component C1 performs a function needed by C2)
  - Timing dependencies (C1 must wait for C2)
  - Resource dependencies (C1 and C2 share hardware), etc.
- Coupling
  - We characterize the set of dependencies between components C1 and C2 as their interface
  - Components with many dependencies are characterized as tightly coupled
- Component interfaces == human interfaces!
  - Where components depend on each other, developers must communicate to build components that interact correctly
  - Higher coupling => more communication to get it right
Common Problems

• Unless we are careful about managing dependencies between work assignments, tend to see coordination problems

• Common kinds of complaints:
  – Site A cannot make progress on its code until site B implements X
  – A change to code being developed at A means code developed at B stops working
  – A small change to code at site A requires large changes to code at several other sites
  – Code at A doesn’t work until code at B works, but code at B doesn’t work until code at C works, but code at C needs code at A
  – Developers at site B cannot test their code until developers at A provide critical functionality
  – A and B’s components work when tested independently but deadlock when integrated

Design Goals

• Design Problem: How to structure the software to minimize the need for coordination and communication?

• Desired Solution: divide the system into parts such that:
  – Each team knows exactly what to build
  – Each team can work on their part independently (without having to know the details of what other teams are doing)
  – Teams can work concurrently
  – A change to the implementation (insides) of one part does not affect other parts
  – The parts work together when integrated
Role of Software Architecture

- Questions about properties of system structure are questions about *software architecture*
- Review: formalize what we mean by “part” and “relationships” in terms of architectural structures
- Then consider some architectural structures important for DSD

Software Architecture

Definitions
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.” *

i.e. components, interfaces and relationships

*From Software Architecture in Practice, Bass, Clements, Kazman

Examples

- Different architectural structures
  - Software components
  - Component interfaces
  - Relationships among them

- Examples

<table>
<thead>
<tr>
<th>Structure</th>
<th>Components</th>
<th>Interfaces</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls Structure</td>
<td>Programs (methods, services)</td>
<td>Program interface and parameter declarations</td>
<td>Invokes with parameters (A calls B)</td>
</tr>
<tr>
<td>Data Flow</td>
<td>Functional tasks</td>
<td>Data types or structures</td>
<td>Sends-data-to</td>
</tr>
<tr>
<td>Process</td>
<td>Sequential program (process, thread, task)</td>
<td>Scheduling and synchronization constraints</td>
<td>Runs-concurrently-with, excludes, precedes</td>
</tr>
</tbody>
</table>
Implications of the 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.” - Bass, Clements, Kazman

- Systems have more than one architecture
  - There is more than one useful decomposition into components and relationships
  - Each characterizes kinds different system properties
- It exists whether any thought goes into it or not
  - Decisions are necessarily made if only implicitly
  - Important control issue is who makes them and when

Fit in the Development Cycle

“...The earliest artifact that enables the priorities among competing concerns to be analyzed, and it is the artifact that manifests the concerns as system qualities.”

Why are the early design decisions important to get right?
Effects of Architectural Decisions

- System and development properties affected by the system structure(s)
- System run-time properties
  - Performance, Security, Availability, Reliability, Concurrency
- System static properties
  - Modifiability, Portability, Reusability, Testability
- Production properties (effects on project)
  - Work Breakdown Structure, Scheduling, Time to market
- Business/Organizational properties
  - Lifespan, Versioning, Interoperability, Target market
- Which properties are not established in architecture?

Which structures should we use?

- Choice of architectural views applied reflects the choice of design goals
- Compare to architectural blueprints
  - Different blueprint for load-bearing structures, electrical, mechanical, plumbing
- Each is a view of the same building
- Different views answer different kinds of questions
  - How many electrical outlets are available in the kitchen?
  - What happens if we put a window here?
Models/Views

- Designing for particular software qualities requires the right architectural model or “view”
  - Any model can present only a subset of system structures and properties (an abstraction)
  - Different models allows us to answer different kinds of questions about system properties
  - Need a model that makes the properties of interest and the consequences of design choices visible to the designer, e.g.
    - Process structure for run-time property like performance
    - Module structure for development property like maintainability

Architectures of Interest

- Module structure
  - Specifies the design-time decomposition of the program into work assignments
  - Useful for: separating concerns, ease of change, work breakdown
- Uses structure
  - Specification of the inter-program dependencies
  - Useful for: sub-setting, incremental development
- Process structure
  - Decomposition of the run time structure (tasks, threads)
  - Useful for: controlling run-time dependencies, concurrent execution
- Also: Data Structure, Deployment (physical), etc.
Interfaces

• An interface is a boundary across which components interact
• A component’s interface is *view specific*
  – Elements show up in more than one view
  – Show only aspects of interest in that view
  – E.g., calls view of a module vs. data flow view
• Documenting an interface consists of:
  1. Identifying it (naming)
  2. Documenting syntactic information (e.g. signature)
  3. Documenting semantic information (behavior)

Designing Architecture for Distributed Teams

Module Structure
Work-Breakdown Goals

Goal: decompose the work into tasks requiring the least inter-site communication (in general)

• During development
• When requirements change

Q: Properties should the components have?

DSD Architectural Design Goals

• Limit the necessity for communication by limiting the dependencies (coupling) between components
• Goals: divide the system into work assignments such that
  – Each part can be assigned to a different team and developed and verified independently
  – It is possible to change the implementation details of one module without affecting other modules
  – Only properties of the system that are unlikely to need to be change are used by other modules
  – Works when put back together
In practice this means…

- Want to decompose into components that
  - Are team work assignments
  - Implement all requirements
- Inter-component dependencies are minimized
  - Expose only what is necessary
  - Encapsulates properties likely to change
- Services provided by each component are well defined

Information-Hiding Structure

- Architectural model: called the “information hiding” structure
- Components
  - Called modules
  - Leaf modules are work assignments
- Relations
  - “submodule-of”
  - The set of submodules of any module X's functionality
  - Constrained to be acyclic tree (hierarchy)
- Module interfaces
  - Modules at the leaves of the tree provide the methods implementing the system's functionality
  - Module interfaces specify everything one must know to use the module's services correctly
  - Modules encapsulate properties other modules should not depend on
What is a module?

- Goal: divide the software into independent work assignments. Each work assignment is called a "module."
- A module is characterized by two things:
  - Its interface: services that the module provides to other parts of the system
  - Its secrets: what the module hides (encapsulates). Design/implementation decisions that other parts of the system should not depend on
- Modules are abstract, design-time entities
  - Modules are "black boxes" – specifies the visible properties but not the implementation
  - May or may not directly correspond to programming components like classes/objects
A Simple Module

- A simple integer stack
- The *interface* specifies what a programmer needs to know to use the stack correctly, including:
  - *push*: push integer on stack top
  - *pop*: remove top element
  - *peek*: get value of top element
  - What else?
- The *secrets* (encapsulated) any details that might change from one implementation to another
  - Data structures, algorithms
  - Details of class/object structure
- Defined by a *Module Interface Specification*

Interface Specification

*Module Interface Specifications*
- Documents all assumptions user’s can make about the module’s externally visible behavior (of leaf modules)
  - Access programs, events, types, undesired events
  - Design issues, assumptions
  - Concurrency and timing constraints
- Document purpose(s)
  - Provide all the information needed to write a module’s programs or use the programs on a module’s interface (programmer’s guide, user’s guide)
  - Specify required behavior by fully specifying behavior of the module’s access programs
  - Define any constraints
  - Define any assumptions
  - Records design decisions
Why these properties?

**Module Implementer**
- The specification tells me exactly what capabilities my module must provide to users
- I am free to implement it any way I want to
- I am free to change the implementation if needed as long as I don’t change the interface

**Module User**
- The specification tells me how to use the module's services correctly
- I do not need to know anything about the implementation details to write my code
- If the implementation changes, my code stays the same

---

Key idea: the abstract interface specification defines a contract between a module’s developer and its users that allows each to proceed independently

---

What is an abstract interface?

- An *abstract interface* defines the set of assumptions that one module can make about another
- While detailed, an abstract interface specification does not describe the implementation
  - Does not specify algorithms, private data, or data structures
  - Preserves the module’s secrets
- One-to-many: one abstract module specification allows many possible implementations
  - Developer is free to use any implementation that is consistent with the interface
  - Developer is free to change the implementation
A method for specifying abstract interfaces

1. Define services provided and services needed (assumptions)
2. Decide on syntax and semantics for accessing services
3. In parallel
   - Define access method syntax (signatures)
   - Define access method effects
   - Define terms and local data types
   - Define visible module states
   - Record design decisions
   - Record implementation notes
4. Define test cases and use them to verify access methods
   - Cover testing effects, parameters, exceptions
   - Design test cases before implementing module
   • Can use Javadoc or similar

Quick Example: FWS Data Banker

Sensor Monitor → Data Banker → Message Generation → Transmitter

Producer Tasks → Buffer → Consumer Tasks
An FWS Example: The Data Banker Interface Specification

Define services provided

<table>
<thead>
<tr>
<th>Service</th>
<th>Provided By</th>
<th>Tested By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initialize the set of stored sensor readings.</td>
<td>initialize</td>
<td>TC1, TC2, TC3, TC4, TC5</td>
</tr>
<tr>
<td>2. Store a new sensor reading, maintaining only the necessary history, and retrieve the current sensor reading history, keeping reads and writes synchronized.</td>
<td>read, write</td>
<td>TC1, TC2, TC3, TC4, TC5</td>
</tr>
</tbody>
</table>

Decide on syntax and semantics for accessing services

**Access Methods**

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Parameter name</th>
<th>Parameter type</th>
<th>Description</th>
<th>Exceptions</th>
<th>Map to services</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize</td>
<td>sensorType</td>
<td>String</td>
<td>Type of sensor.</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>sensorType:I r:I</td>
<td>String SensorReading</td>
<td>Type of sensor. Sensor reading value</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>read:0</td>
<td>sensorType:I o:O</td>
<td>String Vector&lt;SensorReading&gt;</td>
<td>Type of sensor. Vector of elements of type SensorReading</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
An FWS Example: The Data Banker Interface Specification

Decide on syntax and semantics for accessing services

**Access Method Semantics**
- Values returned
- State changes
- Legal call sequences
- Synchronization and other call interactions

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize</td>
<td>Initializes a vector of elements of type sensorType of length HistoryLength for each sensor of sensorType with initial values of null</td>
</tr>
<tr>
<td>write</td>
<td>Adds the SensorReading r to the back of the queue and removes the oldest sensor reading value from the front of the queue.</td>
</tr>
<tr>
<td>read</td>
<td>Returns the vector of sensor readings of type sensorType. With the most recent values of the sensor readings. The vector is of length (HistoryLength * number of sensors of that type).</td>
</tr>
</tbody>
</table>

**Synchronization:** This module supports concurrent access to the read and write methods. Where any read or write can occur concurrently, the read and write statements act as atomic operators (i.e., the user will see either the sequence read.write or the sequence write.read).

---

An FWS Example: The Data Banker Interface Specification

- Decide on syntax and semantics for accessing services
- Local Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>HistoryLength</td>
<td>The number of sequential, past sensor values kept</td>
</tr>
</tbody>
</table>

- and Types Used

<table>
<thead>
<tr>
<th>Type</th>
<th>Value Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>SensorReading</td>
<td>A triple (r, v, w) where r is of type SensorReading.resolution, v is of type SensorReading.value, and w of type SensorReading.weight</td>
</tr>
</tbody>
</table>
Define test cases and use them to verify access method

**Example**

### 1.1.1 T1

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Input Type/Value</th>
<th>Expected Results</th>
<th>Service</th>
<th>Preamble</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initialize</td>
<td>sensorType</td>
<td>Type of sensor.</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>read</td>
<td>sensorType</td>
<td>Returns vector of null values</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

**Interface Design Issues**

1. Should we let the user read an empty vector of sensor readings after initialization, or just throw an exception?

A1. Yes. An empty vector should be treated just as any other.
A2. No. There are no valid values in an empty vector that can be averaged, so we should let the user know that the vector is empty by throwing the exception.

Resolution: Yes. We will check values during testing to save space and CPU cycles.
### Interface Design

#### Design Principles

Considerations in Interface Design

Role of Information Hiding and Abstraction
Three Key Design Principles

- Address the basic issue: which constructs are essential to the problem solution vs. which can change
  - “Fundamental assumptions”
  - “Likely changes”
- Most solid first
- Information hiding
- Abstraction

Principle: Most Solid First

- View design as a sequence of decisions
  - Later decisions depend on earlier
  - Early decisions harder to change
- Most solid first: in a sequence of decisions, those that are least likely to change should be made first
- Goal: reduce rework by limiting the impact of changes
- Application: used to order a sequence of design decisions
  - Generally applicable to design decisions
  - Module decomposition – ease of change
  - Developing families – create most commonality
Information Hiding

- Information hiding: Design principle of limiting dependencies between components by hiding 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.

Abstraction

- General: disassociating from specific instances to represent what the instances have in common.
  - Abstraction defines a one-to-many relationship. E.g., one type, many possible implementations.
- Modular decomposition: Interface design principle of providing only essential information and suppressing unnecessary detail.
Abstraction

- Two primary uses
- Reduce Complexity
  - Goal: manage complexity by reducing the amount of information that must be considered at one time
  - Approach: Separate information important to the problem at hand from that which is not
    - Abstraction suppresses or hides "irrelevant detail"
    - Examples: stacks, queues, abstract device
- Model the problem domain
  - Goal: leverage domain knowledge to simplify understanding, creating, checking designs
  - Approach: Provide components that make it easier to model a class of problems
    - May be quite general (e.g., type real, type float)
    - May be very problem specific (e.g., class automobile, book object)

Example: Simple Library Model

- What are the abstractions?
- What information is hidden?
Applying Principles

• Most-solid: asks the questions
  – Which design decisions are least likely to change?
  – Equivalently: would this decision be true of any implementation we will do? Would it be common to all applicable implementations?

• Information hiding
  – Are these design decisions likely to change together or independently?

• Abstraction
  – Which capabilities are essential to the problem that must be solved?
  – Which details are irrelevant to the problem?

Module Interface Design Goals

• General design goals addressed by module interface design
  – Support architectural goals: Independent work assignments, maintainability, understandability, testability, etc.
  – Addressed by two module interface design goals

1. Control dependencies
  – Encapsulate anything other modules should not depend on
  – Hide design decisions and requirements that might change (data structures, algorithms, assumptions)

2. Provide services
  – Provide all the capabilities needed by the module’s users
  – Provide only what is needed (complexity)
  – Provide problem appropriate abstraction (useful services and states)
  – Provide reusable abstractions
1. Controlling Dependencies

- Addressed using the information hiding principle
  - Design principle: limit dependencies between modules by encapsulating information other modules should not depend on
- When thinking about what to put on the interface
  - Design the module interface to reveal only those design decisions considered unlikely to change
  - Required functionality allocated to the module and considered likely to change must be encapsulated
  - Each data structure is used in only one module
  - Data structures may be accessed by programs within the module but not those outside the module
  - Any other program must access internal data by calling access programs on the interface
- Origin of good OOD principles

2. Provide Services

- Interface provides the capabilities of the module to other modules in the system, addressed by:
  - Abstraction: provide only essential information and suppress unnecessary detail
  - Most solid: provide only information that will be true of any implementation of the module
Which Principle to Use

- Use abstraction when the issue is what should be on the interface (form and content)
  - How can I provide the necessary services without unnecessary detail?
  - How can I provide services that are easy to understand and use for the problem?
- Use information hiding when the issue is what information should not be on the interface
  - Which information should users avoid depending on?
  - What is likely to change or do I want to be able to change?

Example: Face Recognition Module
Basic Application

1. Take a picture of John Smith
2. The system searches a pre-established data-base of faces
3. If recognized, retrieves relevant information

Architectural Concept

• Take advantage of existing web services
• Support a family of possible applications
• How should the face recognition module be designed?
**Problem**

- Application security will be provided by face recognition
- Assumption: we will use a web-based face-recognition service (Face++, ReKognition, etc.)
- Problem: design a module to provide face recognition services to the rest of the system
- Design goals: support distributed development, ease of change
- Interface design
  - What should be hidden?
  - What services should be provided?

**Simple Face Recognition Module**

Define services provided

1 **Introduction**

The Simple Face Recognition Module provides the minimal services needed to locate and identify faces in pictures.

This module provides an abstract face recognition service. Its secret is the specific face recognition services used. This includes the face recognition algorithm, data base structures, and whether the algorithm is implemented in hardware or software.

2 **Services**

2.1 **Services Provided**

<table>
<thead>
<tr>
<th>Service</th>
<th>Provided By</th>
<th>Tested By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Train the system to recognize an individual's face in an image.</td>
<td>train</td>
<td></td>
</tr>
<tr>
<td>2. Recognize an individual in a photo and returns the identifier associated with the individual.</td>
<td>recognize</td>
<td></td>
</tr>
</tbody>
</table>
Simple Face Recognition Module

Decide on syntax and semantics for accessing services

Access Methods

2.2 Access Method

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Parameter name</th>
<th>Parameter type</th>
<th>Description</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>train</td>
<td>identifier:1</td>
<td>String</td>
<td>Individual identifier</td>
<td>noPhoto, noFace</td>
</tr>
<tr>
<td></td>
<td>photo:1</td>
<td>JPEG</td>
<td>JPG photo link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>trained:0</td>
<td>Boolean</td>
<td>False if training failed</td>
<td></td>
</tr>
<tr>
<td>recognize</td>
<td>photo:1</td>
<td>URL</td>
<td>JPG photo link</td>
<td>noPhoto, noFace</td>
</tr>
<tr>
<td></td>
<td>identifier:0</td>
<td>String</td>
<td>Individual identifier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reliability:0</td>
<td>Percent</td>
<td>Certainty of recognition in percent</td>
<td></td>
</tr>
</tbody>
</table>
2.3 Access Method Effects

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>train</td>
<td>If the face recognition system detects a face in the photo, associates the facial features with the given identifier. Parameter trained returns false if the call does not result in any training (training fails). In general, each successful training increases the recognition reliability for the associated individual. Assumption: expects a single face in the image, otherwise, results are indeterminable.</td>
</tr>
<tr>
<td>recognize</td>
<td>If the face recognition system associates a face in the photo with a particular identifier, returns that identifier. Else, if the face recognition system associates a face in the photo with more than one identifier, returns the identifier with the highest certainty of recognition. reliability returns the degree of certainty of recognition as a percent where 100 represents certainty and 0 means that the face could not be recognized. Assumption: expects a single face in the image, otherwise, results are indeterminable.</td>
</tr>
</tbody>
</table>

The system cannot guarantee the results of training the system on multiple photos of an individual. However, in general, for a given identifier, i and set of distinct facial photos \((p_{i,1}, \ldots, p_{i,n})\) of the individual associated with \(i\) for the two sequences of calls:

\[
\text{train}(i, p_{i,1}, \ldots, \text{train}(p_{i,n}), \text{recognize}(p_{i,1}, t_i), \ldots, \text{train}(p_{i,n}), \text{recognize}(p_{i,1}, t_i))
\]

if \(t_i = \text{true}\) for every call then \(t_i > t_i\). That is, the training on additional photos generally increases the reliability of recognition.

An FWS Example: The Data Banker Interface Specification

Local Data Types

3 Local Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>percent</td>
<td>integer(0,100)</td>
</tr>
</tbody>
</table>

and Terms

4 Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>training</td>
<td>The process of associating facial features with an individual identifier by supplying the system different photos of that individual.</td>
</tr>
<tr>
<td>Recognition</td>
<td>The process of associating the facial features in a given photo with features of individuals on which the system has been trained.</td>
</tr>
</tbody>
</table>
Simple Face Recognition Module

Record design decisions

6 Interface Design Issues
The goal of interface design is to abstract the common capabilities that should be available from any facial recognition system and provide an abstract face recognition service.

1. Should the interface support identification of multiple faces in a photo?
   a. Yes. Many photos will have multiple faces and this capability is supported by at least one service
   b. No. What is known is that every face recognition service necessarily supports the recognition of a single face. In addition, we need a simple interface for feasibility testing.
   Resolution: The decision was made to provide the simplest possible service first and extend it later if necessary.

2. 

Simple Face Recognition Module

Record active review questions

8 Questions for Reviewers
1. Assume i is an individual’s identifier and p is a photo that includes i’s face. If the first call to this module is \texttt{recognize}(i, p), what value of r should be returned?
2. Assume i is an individual’s identifier and p is a photo that includes i’s face. What should be the result of the call \texttt{recognize}(p, i)?
   a. If the facial recognition system cannot associate the face with any individual it has been trained on?
   b. If the system recognizes the face with a 27% certainty?
   c. If the system cannot detect any face in the photo?
Simple Face Recognition Module

Define test cases and use them to verify access method

Example

T1 Normal expected:
<with a face that is not in the database, photos a...k, m of the same person>
recognize(m) (expect no match or a low confidence match)
train(a); train(b); ... train(k); (or a single call to train with a collection of photos)
recognize(m) (expect a high confidence match)
recognize(n) where n is different from the database. Expect no match or a low confidence match

Questions?
Example: Car Object

- What are the abstractions?
  - Purpose of each?
- What information is hidden?