Attribute-Driven Design Method

April 2015

Ying SHEN

SSE, Tongji University

Lecture objectives

This lecture will enable student to

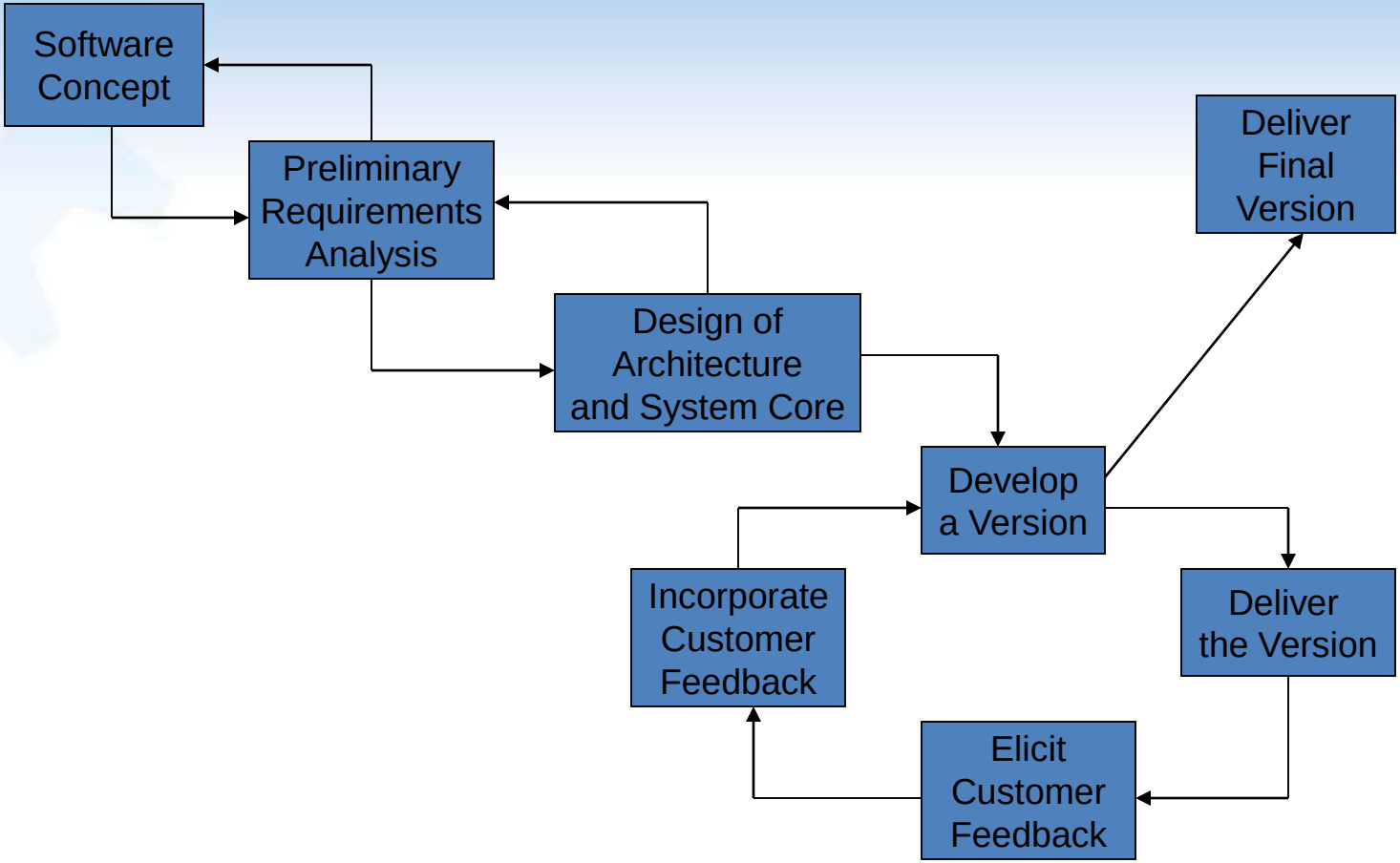
- understand ADD steps
- design the architecture using ADD method

Architecture in the life cycle

Evolutionary Delivery Life Cycle (EDLC)

- User and customer feedback
- Iterate through several releases before final release
- Adding of functionality with each iteration

Evolutionary delivery life cycle



When do we start developing the SA?

Requirements come first

- But not all requirements are necessary to get started

Architecture shaped by (remember the ABC)

- Functional requirements
- Quality requirements
- Business requirements
- Expertise and experience of architects

We call these requirements “architectural drivers”

Determining architectural drivers

To identify the Architectural Drivers

Identify the highest priority Business Goals

- Only a few of these

Turn these into scenarios or use cases

Choose the ones that have the most impact on the architecture

- These are the architectural drivers
- There should be less than 10 of these

Attribute driven design (ADD) method

Method to design architecture so that both functional and quality requirements are met

Defines SA by decomposing based on the quality attributes

Recursive decomposition process; at each stage

- Tactics are chosen to satisfy some qualities
- Functionality is added

ADD input

Input:

1. The set of quality scenarios for drivers
   - Key drivers may change during design, due to
     - better understanding or changing of requirements
   - Quality attribute requirements are a good start
     - although they can't be all known *a priori*
2. The context description
   - *What are the boundaries of the system being designed?*
   - *What are the external systems, devices, users, and environmental conditions with which the system being designed must interact?*

ADD output

Output: a set of sketches of architectural views.

- First several levels of a module decomposition view of an architecture
- Not all details of the views result from applying ADD
- System described as
  - a set of containers for functionality
  - the interactions among the containers

ADD output

Critical for achieving desired qualities

Provides framework for achieving the functionality

Difference between ADD output and an architecture ready for implementation

- Detailed design decisions postponed
  - e.g. choosing the names and parameter types of interface
- Flexibility

Case study: Garage Door Opener

Garage door opener: A real-time system responsible for raising and lowering the garage door, via

- Switch button
- Remote control
- Home information system

It is possible to diagnose problems of opener from the home information system (HIS)

Product line architecture! (PLA)

- The processor is different from other products.

Case study: Garage Door Opener

Input to ADD: a set of requirements

- Functional requirements as use cases
- Constraints
- Quality requirements
  - Set of system-specific quality scenarios
  - Only the necessary detail

Case study: Scenarios

Scenarios for garage door system

- Reacting to obstacles [Performance]
  - If an obstacle (person/object) is detected by garage door during descent, *it must halt or re-open within 0.1 second* and report to HIS and user interface.
- Door commands [Performance]
  - Remote control; HIS; button.
  - Open, close, halt, diagnosis
  - Detect a command and initiate execution within 0.5 sec

Case study: Scenarios

Scenarios for garage door system

- Processors [Modifiability]
  - Processors can change due to either obsolescence or changes in the marketplace.
- UI [Modifiability]
  - Various garage door openers have various controls.
- Garage door opener should be accessible for diagnosis and administration from within the HIS
  - Using a product-specific diagnosis protocol

ADD steps

Steps involved in Attribute Driven Design (ADD)

1. Choose the module to decompose
   - Start with entire system
   - Inputs for this module need to be available
     - Constraints, functional and quality requirements

ADD steps

2. Refine the module
   - Choose architectural drivers
   - Choose architectural pattern
   - Instantiate modules and allocate functionality using multiple views
   - Define interfaces of child modules
   - Verify and refine use cases and quality scenarios and make them constraints for child modules

3. Repeat for every module that needs further decomposition

Choose the module to decompose

Modules: system → subsystem → submodule

Current design element is the whole system

Constraint for the Garage Door Opener system

- Garage door opener is the system
- One constraint: opener must interoperate with the home information system (HIS)

ADD steps

1. Choose the module to decompose
2. Refine the module
   - Choose architectural drivers
   - Choose architectural pattern that satisfies these drivers
   - Instantiate modules and allocate functionality
   - Define interfaces of child modules
   - Verify and refine use cases and quality scenarios and make them constraints for child modules
3. Repeat for every module that needs further decomposition

Refine the module: Choose arch. drivers

Architectural drivers: functional and quality requirements that shape the architecture

- Among the top priority requirements for the module
- To be addressed in the initial decomposition of the system

Drivers for Garage Door Opener system

- Real-time performance
- Modifiability to support product lines
- Online diagnosis supported

ADD steps

1. Choose the module to decompose
2. Refine the module
   - Choose architectural drivers
   - Choose architectural pattern that satisfies these drivers
   - Instantiate modules and allocate functionality
   - Define interfaces of child modules
   - Verify and refine use cases and quality scenarios and make them constraints for child modules
3. Repeat for every module that needs further decomposition

Refine the module: Choose arch. pattern

For each quality requirement there are

- identifiable tactics and patterns to implement them
- each tactic is designed to realize one/more attributes
- the patterns in which the tactics are embedded have impact on other attributes
- balance between qualities needed
  - As composition of tactics is used

We need to identify child modules required to implement the tactics

Refine the module: Choose arch. pattern

The goal of this step is

- to establish an overall architectural pattern for the module
- Architectural pattern should satisfy drivers
- Architectural pattern is built by “composing” the tactics selected
- Two factors involved in selecting tactics:
  - Architectural drivers themselves
  - Side effects of the pattern implementing the tactic on other requirements

Refine the module: Choose arch. pattern

Example 1: Modifiability at design time

- Increase semantic coherence and information hiding
  - Separate responsibilities dealing with user interface, communication, sensors, diagnosis into their own module
    - These modules are called virtual machine.
  - The first 3 virtual machines: will vary for the different products to be derived from the PLA

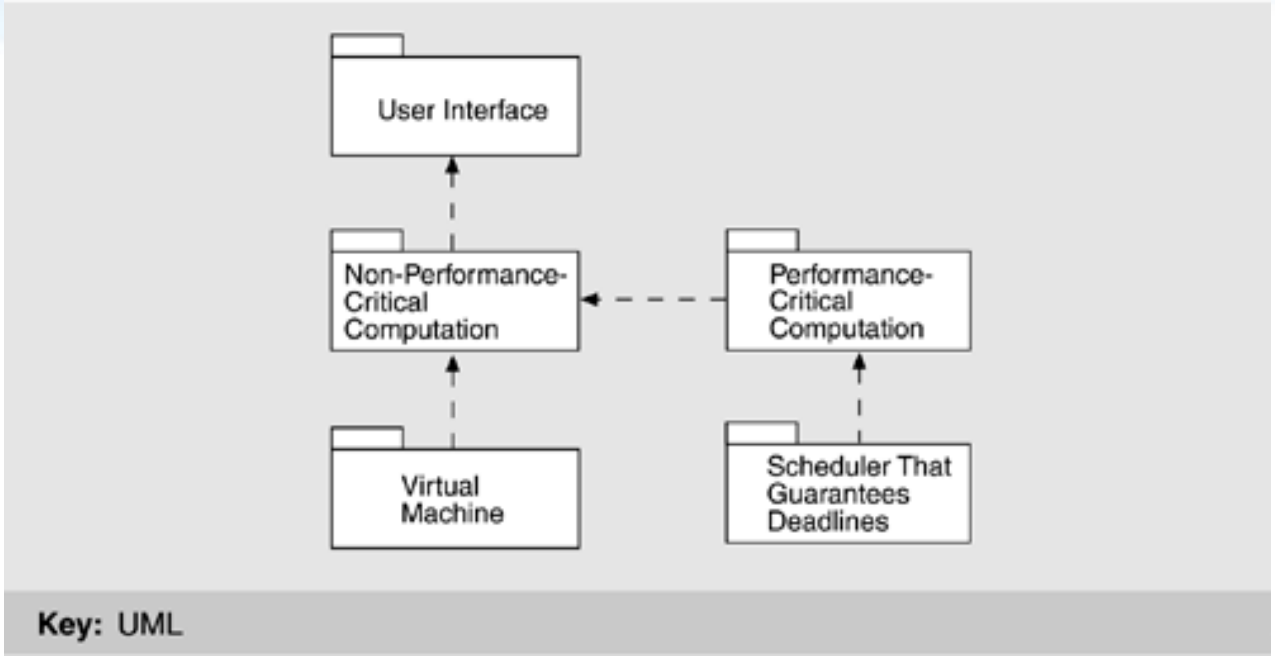
Refine the module: Choose arch. pattern

Example 2: Performance

- Increase computational efficiency and choose scheduling policy
  - Performance-critical computations made efficient
  - Performance-critical computations scheduled to achieve the timing deadline

Refine the module: Choose arch. pattern

Architectural patterns that utilize tactics to achieve garage door drivers



ADD steps

1. Choose the module to decompose
2. Refine the module
   - Choose architectural drivers
   - Choose architectural pattern that satisfies these drivers
   - Instantiate modules and allocate functionality
   - Define interfaces of child modules
   - Verify and refine use cases and quality scenarios and make them constraints for child modules
3. Repeat for every module that needs further decomposition

Refine the module: Instantiate modules

Previous step

- We defined the module types of the decomposition

Now

- We instantiate them

Virtual machine manages communication and sensor interactions

- Software running on top of VM: an application
- One module for each group of functionality – instance of the module types

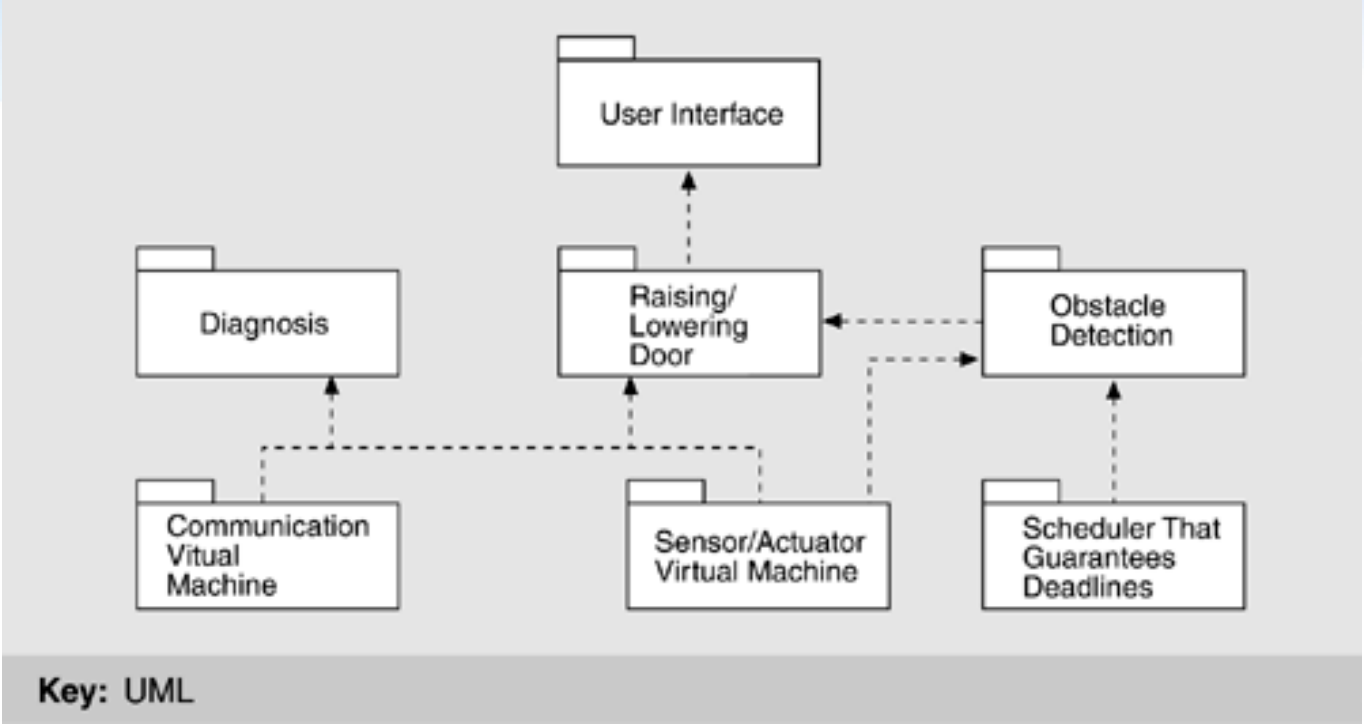
Refine the module: Instantiate modules

Our example

- Responsibility for managing obstacle detection and halting door → performance critical section
  - This functionality has deadline
- Management of normal door raising/lowering
  - No deadline => non-performance critical section
  - Same with diagnosis
- Several responsibilities for VM
  - Communication and sensor reading and actuator control
  - Results in 2 instances of VM

Refine the module: Instantiate modules

First level decomposition of SGDO



Refine the module: Allocate functionality

- Use case for parent module → understand distribution of functionality
- Add/remove child modules → to fulfill required functionality
- Every use case of parent module → representable by a sequence of responsibilities within child modules

Refine the module: Allocate functionality

User interface

- Recognize user requests and translate them into the form expected by the raising/lowering door module.

Raising/lowering door module

- Control actuators to raise or lower the door. Stop the door when it reaches either fully open or closed.

Obstacle detection

- Recognize when an obstacle is detected and either stop the descent of the door or reverse it.

Refine the module: Allocate functionality

Communication virtual machine

- Manage all communication with HIS.

Sensor/actuator virtual machine

- Manage all interactions with the sensors and actuators.

Scheduler

- Guarantee that the obstacle detector will meet its deadlines.

Diagnosis

- Manage the interactions with HIS.

ADD steps

1. Choose the module to decompose
2. Refine the module
   - Choose architectural drivers
   - Choose architectural pattern that satisfies these drivers
   - Instantiate modules and allocate functionality
   - Define interfaces of child modules
   - Verify and refine use cases and quality scenarios and make them constraints for child modules
3. Repeat for every module that needs further decomposition

Refine the module: Define interfaces of child modules

Module interface

- Services and properties provided and required
- Documents what others can use and on what they can depend

Module decomposition view documents

- Producers/consumers on information
- Patterns of interaction requiring modules to provide and use services

Refine the module: Define interfaces of child modules

User interface:

- openDoor: send command to Raising/lowering door module to open garage door
- closeDoor: to close door
- haltDoor: to halt door (manually halt function is available)

Raising/lowering door module

- startOpen: send command to Sensor/actuator virtual machine module to turn on the actuator and open door.
- startClose: turn on the actuator and close door.
- stop: turn off the actuator and left the door in the current position

ADD steps

1. Choose the module to decompose
2. Refine the module
   - Choose architectural drivers
   - Choose architectural pattern that satisfies these drivers
   - Instantiate modules and allocate functionality
   - Define interfaces of child modules
   - Verify and refine use cases and quality scenarios and make them constraints for child modules
3. Repeat for every module that needs further decomposition

Refine the module: Verify and refine

Decomposition into modules needs to be verified.

Child modules need preparation for their own decomposition on second level decomposition.

Done for

- Functional requirements
- Constraints
- Quality requirements

Functional requirements

Child modules have responsibilities

- Derived from functional requirements decomposition
- As use cases for the module
- Split and refine parent use cases
- Example: use cases that initializes the whole system
  - Broken into initialization of subsystems

Functional requirements

Case study:

Initial responsibilities

- Open/close door on request
  - Locally or remotely
- Stop door when detect obstacle
- Interact with HIS
- Support remote diagnostics

Functional requirements

Case study:

Decomposition of responsibilities

- User interface:
  - recognize user requests and translate them into form expected by the raising/lowering door module
- Raising/lowering door module:
  - Control actuators to raise/lower door
  - Stop door when fully closed or fully open
- Obstacle detection

Functional requirements

Case study:

Decomposition of responsibilities more

- Communication VM
  - Manage communication with HIS
- Sensor/actuator VM
  - Manage interaction with sensors and actuators
- Scheduler
  - Guarantee deadline
- Diagnosis
  - Manage interaction with HIS for diagnosis

Constraints

Constraints of parent module satisfied by

1. Decomposition satisfies the constraints
   - Using certain OS
2. Constraint satisfied by single child module
   - Special protocol
3. Constraint satisfied by multiple child modules
   - Web usage requires two modules (client and server)

Constraints

Case study:

Constraint: communication with HIS is maintained

- Communication virtual machine will recognize if this is unavailable
- Constraint satisfied by a single child

Quality scenarios (QS)

Need to be refined and assigned to child modules

Possibilities

- QS completely satisfied by decomposition
- QS may be satisfied by decomposition with constraints on child modules
  - Layers and modifiability
- Decomposition neutral with respect to QS
  - QS assigned to child modules
- QS not satisfiable by decomposition
  - Decomposition reconsidered or reason for this recorded
  - Typical trade-offs

Quality scenarios (QS)

Case study:

Device and controls for opening and closing the door are different for the different products in the product line

- May include controls from within the HIS
- QS is delegated to user interface module

Processor used in different products will differ

- Product architecture for each specific processor should be directly derivable from the PLA
- QS is delegated to all modules

Quality scenarios (QS)

Case study:

If an obstacle (person/object) is detected by garage door during descent, it must halt or re-open within 0.1 second

- QS delegated to scheduler and obstacle detection module

Garage door opener should be accessible for diagnosis and administration from within the HIS

- Using a product-specific diagnosis protocol
- QS split between diagnosis and communication modules

Step outcome

Decomposition of module into children

- Each child has
  - Set of responsibilities
  - Set of use cases
  - Interface
  - Quality scenarios
  - Collection of constraints
- Enough for next iteration of decomposition

ADD steps

1. Choose the module to decompose
2. Refine the module
   - Choose architectural drivers
   - Choose architectural pattern that satisfies these drivers
   - Instantiate modules and allocate functionality
   - Define interfaces of child modules
   - Verify and refine use cases and quality scenarios and make them constraints for child modules
3. Repeat for every module that needs further decomposition

Iteration progress

Vocabulary of modules and their responsibilities

Variety of use cases and quality scenarios and understood some of their ramifications

Information needs of modules

- Their interactions

Not decided yet

- Communication language, algorithm for obstacle detection, etc

Iteration outcome

We defined enough to allocate work teams and give them charges

- If we design a large system

We can proceed to next iteration and decide on answers for the questions

- If we design small system

Forming the team structure

When modules fairly stable

- They can be allocated to development teams (existing, new)
- Team structure mirror module decomposition structure

Each team creates its own internal work practices (or adopts a system-wide set)

- Bulletin boards
- Web pages
- Naming conventions for files
- Configuration control system

Quality assurance and testing procedures set up for each group

- Coordinate with others

The End