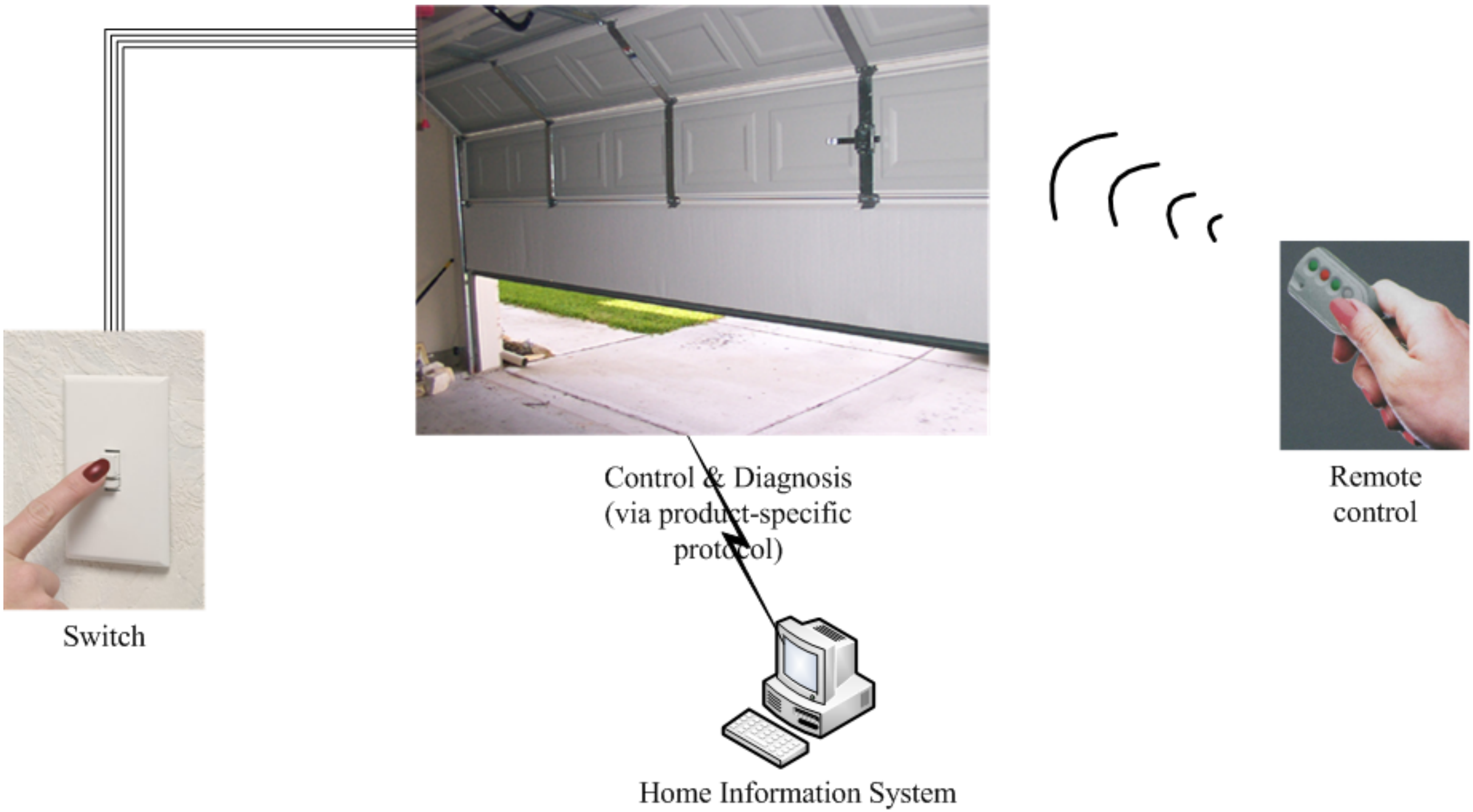
Designing an Architecture (2)

主讲教师：王灿
Email: wcan@zju.edu.cn
TA: 李奇平 liqiping1991@gmail.com
Course FTP: ftp://sa:sa@10.214.51.13

Case Study---Garage Door Opener

- Task: design *a product line architecture* for a garage door opener within a home information system

The Garage Door Opener



ASRs (1)

- The device and controls for opening and closing the door are different *for the various products in the product line*
  - Various devices (various actuators, etc.)
  - Various controls
    - Controls from within a home information system
    - Various switches
    - Various remote controls

  - *The product architecture for a specific set of controls should be directly derivable from the product line architecture*

ASRs (2)

- The ***processor*** used in different products will differ
  - *The product architecture for each specific processor should be directly derivable from the product line architecture*

ASRs (3)

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

ASRs (4)

- The garage door opener should be accessible for diagnosis and administration from within the home information system using ***a product-specific diagnosis protocol***.
  - It should be possible to directly produce an architecture that reflects this protocol

The Steps of ADD

- ADD is a five-step method:
  1. Choose an element of the system to design
  2. Identify the ASRs for the chosen element
  3. Generate a design solution for the chosen element
  4. Inventory remaining requirements and select the input for the next iteration
  5. Repeat steps 1-4 until all the ASRs have been satisfied

Step 2: Identify the ASRs for the Chosen Element

- In our case, requirements to be addressed at this level of design include:
  - Real-time performance requirement *
  - Modifiability requirements
  - Support online diagnose via product-specific protocol
- Requirements are not treated as equals
  - Less important requirements are satisfied within constraints obtained by satisfying more important requirements
    - This is a difference of ADD from other SA design methods

Step 3: Generate a Design Pattern for the Chosen Element (1)

- For each quality requirement there are identifiable tactics to achieve it
  - Or patterns that implement these tactics
- Patterns or tactics usually have an impact on multiple quality attributes.
  - E.g. Using intermediary (modifiability/performance)
- In an architecture design, a composition of many such tactics is used to achieve a balance between the required multiple qualities
  - Achievement of the quality and functional requirements is analyzed during the refinement step

Step 3: Generate a Design Pattern for the Chosen Element (2)

- The goal of this step is to establish an overall architectural ***pattern*** consisting of module types
  - Satisfies the ASRs
  - Constructed by composing selected tactics
- Two factors involved in selecting tactics:
  - ASRs themselves
  - Side effects of a tactic has on other requirements

Step 3: Generate a Design Pattern for the Chosen Element (3)

- Examples:
  - To achieve modifiability, a classic tactic is to use the “Interpreter” pattern (new functions achieved by modifying input---interpreted language)
    - E.g. HTML
  - Note the use of interpreters will have a strong negative effect on performance

  - The decision to use an interpreter depends on the relative importance of modifiability versus performance
    - Or use an interpreter for a portion of the pattern and use other tactics for other portions

Step 3: Generate a Design Pattern for the Chosen Element (4)

- In our garage door opener, we re-examine modifiability tactics
  - Modifiability Tactic categories
    - Reduce size of a module
    - Increase cohesion
    - Reduce coupling
    - Defer binding

  - Since our modifiability scenarios are primarily concerned with design time changes and module size at this point is not an issue → we primarily use "increase cohesion" and "reduce coupling" categories

Step 3: Generate a Design Pattern for the Chosen Element (5)

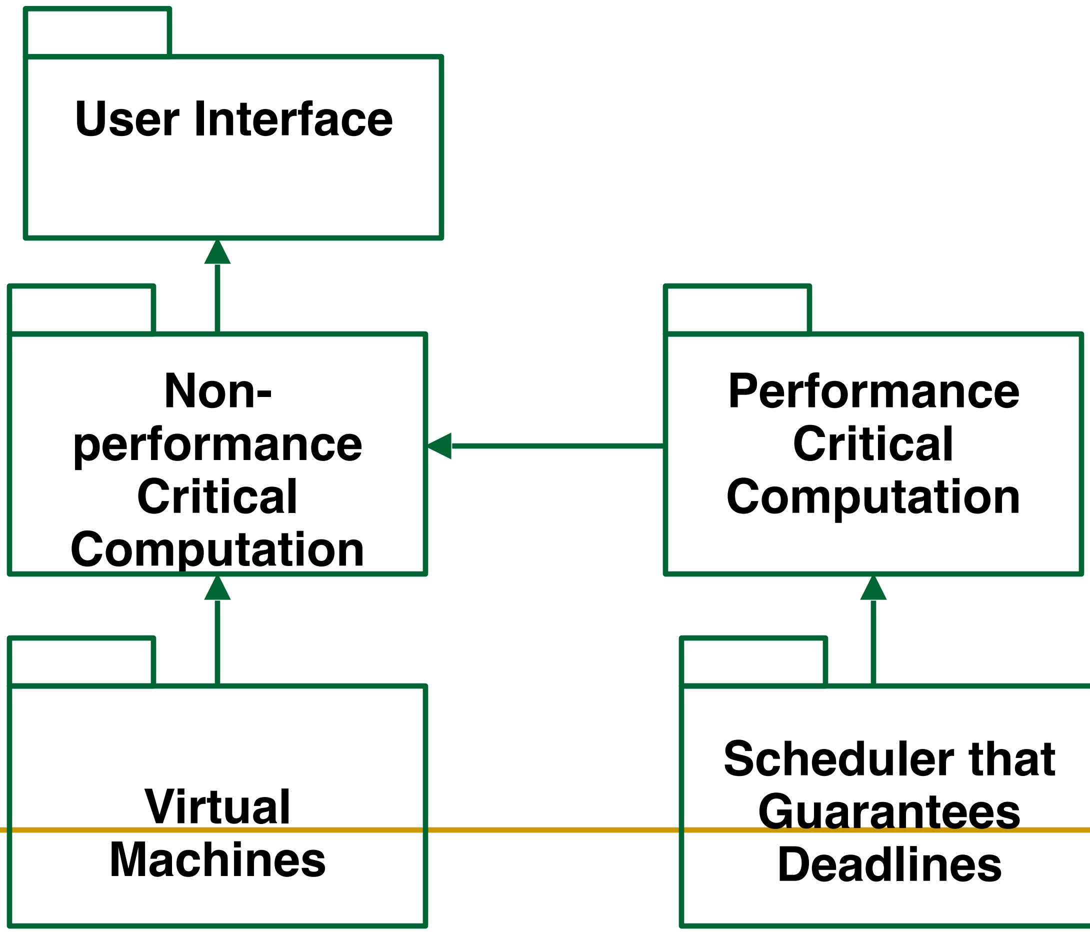
- Re-examine performance tactics
  - Performance Tactic categories
    - Control resource demand
    - Manage resources
  - We will use both “control resource demand” and “manage resources” tactics to achieve the real-time requirement

Step 3: Generate a Design Pattern for the Chosen Element (6)

- we select tactics from above-mentioned categories
  - Increase semantic coherence
    - ***Separate responsibilities dealing with user interface, communication and sensors into their own module***
  - Encapsulate
    - ***"Virtual machines" will be used for communication and sensors because we expect them to vary within product line***
  - Increase resource efficiency
    - ***Improve the algorithm used in the part with real-time requirement***
  - Schedule resources
    - ***Use a scheduler that guarantees deadlines***

Step 3: Generate a Design Pattern for the Chosen Element (7)

- One architectural pattern that utilizes tactics to achieve garage door ASRs



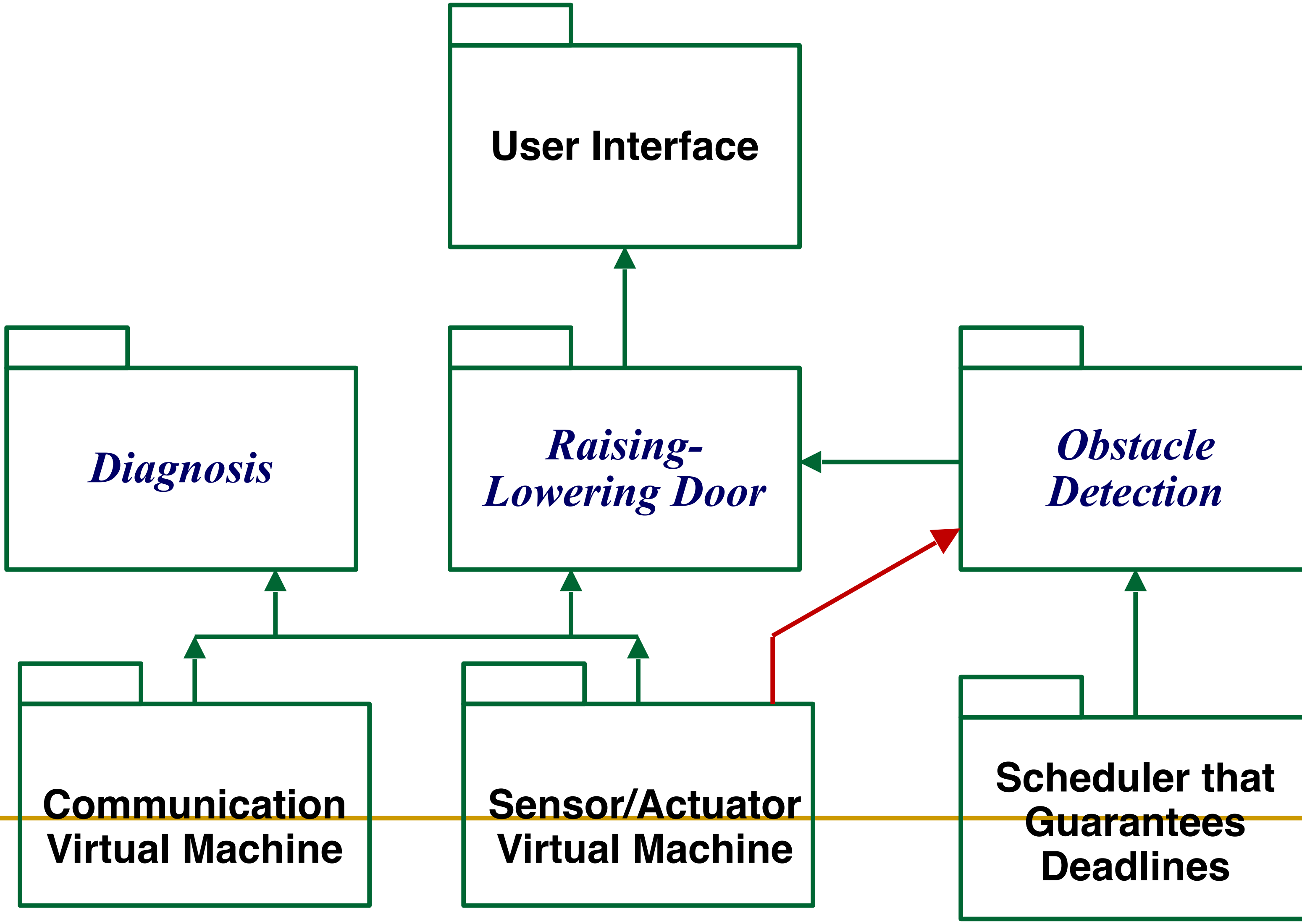
Step 3: First level Decomposition

- User Interface
- Non-performance-critical computation
  - Applications running on top of virtual machine
    - E.g. normal raising/lowering the door
- Virtual machines
  - Managing communication and sensor interactions
- Performance-critical computation
  - e.g., Managing obstacle detection
- Scheduler that guarantees deadlines

Step 3: Allocate Functionality & Instantiate Modules (1)

- The criterion for allocating functionality
  - In a concrete system, we will have one module for each "group" of functionality
  - Theses will be instances of types shown in our pattern
- Functionalities to be provided
  - Raising/lower door (no deadline, non-performance critical)
  - Diagnosis (non-performance critical)
  - Managing obstacle detection (with a deadline, performance-critical)

Step 3: Allocate Functionality & Instantiate Modules (2)



Step 3: Allocate Functionality & Instantiate Modules (3)

- Applying use cases pertaining to the parent module
  - Completeness: every use case of the parent module must be representable by a sequence of responsibilities within the child modules
  - May lead to adding or removing child modules
    - E.g. the scheduler in our example
  - Discovery of necessary information exchange
    - A producer/consumer relationship between those modules
    - Prototype of interface documentation

Step 3: Allocate Functionality & Instantiate Modules (4)

- Some tactics introduce specific patterns of interaction between module types
  - To be recorded, later to be translated into responsibilities for the affected modules
  - E.g. using publish-subscribe style
- The above steps ensure the desired functionalities to be delivered
- What about quality attributes?
  - Will deadlock occur during system execution?
  - Are there data consistency issue?
  - How shall the functionality be deployed?

Step 3: Represent the Architecture with Multiple Views (1)

- At least one view from each viewtype
  - Module decomposition view
    - Allocation of functionality + major data flow among modules
  - Concurrency view
    - Revealing resource contention, deadlock, data consistency issue etc.
    - Likely leading to new responsibilities (to be added in module views)
      - E.g. a resource manager

Step 3: Represent the Architecture with Multiple Views (2)

- To understand concurrency in our example, consider:
  - Two users doing similar things at the same time
    - Resource contention, data integrity
  - One user performing multiple activities simultaneously
    - Data exchange, activity control
  - Starting up & shutting down the system
  - Synchronization in the sensor/actuator virtual machine: performance critical area & raising/lowering door
    - Introduction of a new module: scheduler

Step 3: Represent the Architecture with Multiple Views (3)

- Deployment view
  - Additional responsibilities, multiple instances etc.
  - The derivation of the deployment view and the achievement of quality attributes
    - e.g. replication → performance or reliability
    - e.g. a real-time scheduling mechanism actually prohibit deployment on different processors
- In our example, deployment view presents the problem of dividing responsibility between the door opener system and the HIS
  - E.g. authentication of a remote request

Step 3: Define Interfaces of the Child Modules (1)

- At this level by "Interface to module" we mean document the services and properties provided and required, not the more detailed "signature" of a method.
  - It documents what others can use and on what they can depend.

- Analyzing the decomposition into the three views provides interaction information for the interface
  - Module view
  - Concurrency view
  - Deployment view

Step 3: Define Interfaces of the Child Modules (2)

- The three viewtypes provide:
  - Decomposition view
    - Producers/consumers relations; patterns of interactions
  - Concurrency view
    - Interactions among threads
    - A component is active
    - Synchronization information
  - Deployment view
    - Hardware requirements
    - Timing requirements
    - Communication requirements
- All the above should be available in the modules’ interface documentation

Step 4: Verify and Refine Requirements and Generate Input for the Next Iteration

- Analyzing the previous design “proposal”
  - Verifying the design by “running” the parent’s ASRs
- Preparing the child modules for further decomposition
  - Inheriting use cases/constraints/QAS from the parent

- We will examine this by looking at:
  - Functional requirements
  - Constraints
  - Quality attribute requirements

Step 4: Verify Functional Requirements (1)

- The Garage door system responsibilities can be assigned to the child modules
  - User Interface:
    - handle user requests,
    - translate for raising/lowering module
    - display responses
  - Raising/Lowering door module
    - Control actuators to raise/lower door
    - Stop when completed opening or closing
  - Obstacle detection:
    - Recognize when object is detected
    - Stop or reverse the closing of the door

Step 4: Verify Functional Requirements (2)

- Garage door system responsibilities assigned (continued)
  - Communication virtual machine
    - manage communication with Home Information System (HIS)
  - Sensor/Actuator virtual machine
    - manage interactions with sensors/actuators
  - Scheduler
    - Guarantee that deadlines are met when obstacle is detected
  - Diagnosis:
    - manage diagnosis interaction with HIS

Step 4: Verify Constraints

- In our example, a constraint is to communicate with the HIS using product-specific protocol
  - Delegated to the communication VM

Step 4: Verify Quality Attribute Requirements

- The devices and controls for opening and closing are different in the products of the product line
  - Scenario delegated to user-interface and *sensor/ actuator virtual machine* module
- The processor used in different products will differ
  - Delegated to all modules “don’t use processor specific code”
- If an obstacle is detected, closing must stop within .1 second
  - Delegated to scheduler and obstacle-detection modules

Step 4: What Requirements Are Left?

- Four possibilities that an ASR (the quality attribute requirement, functional requirement, or constraint) is or is not met by the current round of design:
  - Satisfied
    - The design W.R.T. that ASR will is complete, i.e. it will not be considered in the next round of design
  - Delegated to one of the children
  - Distributed among the children
  - Cannot be satisfied with the current design

Summary of ADD

- ADD is a five-step method:
  1. Choose an element of the system to design
  2. Identify the ASRs for the chosen element
  3. Generate a design solution for the chosen element
  4. Inventory remaining requirements and select the input for the next iteration
  5. Repeat steps 1-4 until all the ASRs have been satisfied

Reading Assignment

- Read the case study of the World Wide Web, which has been uploaded to the FTP server