

## Today

Announcement: short (10-pt) HW due tomorrow

Last time:

Architectural Analysis (ref.: chapter 32)

Architecture is very important!

Today:

More on chapter 32 / Review of key points / Leave early

## Architectures – More Basic Vocabulary & Concepts

"Architecture" is a many-dimensional idea!

- Architectural drivers:
  - Non-functional (and functional) requirements that impact architecture
  - A bell/whistle is not an architectural driver
- Architectural factors:
  - Architectural drivers
- Architectural decisions:
  - Define solutions to the architectural drivers, such as:
  - Software structure, removal of requirement, terminate project, etc.

## Architectures – Further Vocabulary & Concepts

- Architectural drivers, architectural decisions, and architectural factors
- Do any refer exclusively to software structures?
  - If so, which?
- Are any synonymous?
  - If so, which?
- Which is/are a consequence of which?

## Architectures and FURPS+

- Recall FURPS+
  - Functionality, usability, reliability, performance, supportability, etc.
  - In the UP framework, each requirement is categorized in a FURPS+ category
  - Which letter goes with which category?
- These help define the factors in a **factor table**
  - (See Table 32.2)
  - UP – factor tables go in the "supplementary specifications" document

## Factor Table and FURPS+

- Recall FURPS+
  - Functionality, usability, reliability, performance, supportability, etc.
  - In the UP framework, each requirement is categorized in a FURPS+ category
- These help define the factors in a **factor table**
  - (See Table 32.2)
  - Which FURPS+ categories are in Table 32.2?

## Cross-Cutting (Non-Functional) Requirements, Low Coupling, and Separation of Concerns

- Recall FURPS+
  - Functionality, usability, reliability, performance, supportability, etc.
- We've spent much time on modularizing functionality
  - ...to achieve e.g. low coupling
- Cross-cutting other requirements mess things up!
- Is the reality of cross-cutting requirements a fatal condemnation of the OO paradigm?

## Cross-Cutting (Non-Functional) Requirements, Low Coupling, and Separation of Concerns

- We'd like to somehow modularize ...
  - security, persistence, logging, etc., ...
  - to reduce coupling of each **functional** module to the overall system
- These non-functional, cross-cutting requirements are also "**concerns**"
- Let's **separate** them from the functional modules (as much as is feasible)
  - This **separation of concerns** idea means that we wish to: minimize coupling between ...
  - ... functional modules and the system

## Techniques for Separation of Concerns

- Method 1: each functional module fends for itself
  - E.g. each writes its own log messages to the log file
  - This is a "bad" design
  - High coupling exists
    - (e.g. all modules must agree on the file name – this is a well-nigh unparadonable sim!)
  - Method 1 is not really a method at all
    - – it's exactly the sort of thing we have fields like software engineering and programming languages ...to help us avoid

## Techniques for Separation of Concerns

- Method 2: wrap functional modules in **containers**
  - ...that handle the cross-cutting concern
- Example: declare various objects (Sale, etc.)...
  - ...as "logging" objects
- Write a "logging" module that handles logging
- The compiler does the rest

## Techniques for **Separation of Concerns**

- Method 3: post-compilation code modification
  - Unbeknownst to the programmer of Sale,
    - Someone else has a file listing the names
      - Sale (and all other functional modules that should log)
  - A post-compiler modifies the compiled Sale code
    - (to do logging)
  - The Sale programmer doesn't even know it!
    - Coupling is reduced from method 2

## Aspect-Oriented Programming

- Aspect-oriented programming (AOP)
- AOP doesn't invalidate OOP...
- ...it adds to it, and may be the next major development in the software engineering paradigm
  - (in order: Structured programming, OO, UML, AOP?)
- After compilation,
  - Cross-cutting concerns are woven into the compiled code
  - Developers of functional modules need not worry about it
- The major example: AspectJ
  - (aspect-oriented Java)
  - See [www.aspectj.org](http://www.aspectj.org)
- The future could hold exciting developments!

## Review

- The insides of the covers (4 "pages") summarize key points!