CS3215: Software Engineering Project

CS3215, LN set #5: SE Principles

1. Abstraction and Information Hiding
2. Separation of Concerns (SoC)
3. Modularity, Decomposition
4. Generality
5. Design for Change
6. Rigor and Formality

Is software complex?

Software may be as complex as these:

Large software systems are among most complex systems built by humans

Do we have right methods and tools to develop and maintain software systems?

People can do amazing things

2000 BC

12th century

20th century

- they are large and complex structures
- they were built without modern technologies!
- it took huge effort and cost to build them
  - took some 27 years to build each pyramid
  - thousands people died building them

We can build large, complex software systems, too!

1. Most businesses today depend on complex software systems
2. IBM OS (1960's), huge, complex
3. Military software is huge, complex, must be reliable
4. WINDOWS (close to 100 million LOC)

It takes enormous effort and entails much risk to develop large, complex software

How can we develop software faster, at lower cost?

Software engineering challenges

despite new technologies and many successes there are problems:

- Software projects are often unpredictable
  - many projects run out of schedule and budget,
  - 25% of large projects are never completed
- Maintenance cost up to 80% of computing cost
- Reuse has not become a standard practice
- Low level of automation
- Outsourcing

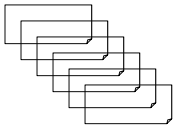
SE principles

1. Abstraction and Information Hiding
2. Separation of Concerns (SoC)
3. Modularity, Decomposition
4. Generality
5. Design for Change
6. Rigor and Formality

SE principles help us manage complexity

1. of a software system (product)
2. of the development process

What it takes to understand code?



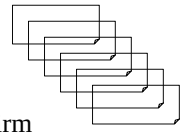
100,000 lines of Java code
in many directories and files

- what does this code do?
- how is code designed?
  - what are roles of various files? what are they for?
  - which files form logical groups?
  - how are files related to each other?
- what is this class for? where is it used?
- what is this variable for? where is it used?

Suppose I tell you what this code does

- This code implements a Facility Reservation Systems (FRS)
- FRS allows us to:
  - reserve facilities such as rooms, equipment, …
  - manage reservations: create, change, cancel, confirm
  - manage FRS users
  - define user rights

FRS code



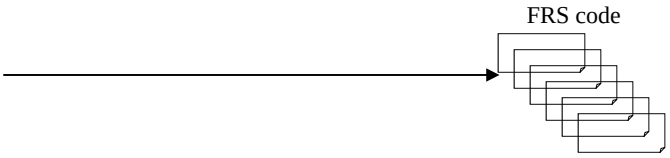
Which principle is in play here?

1st principle: Abstraction

Abstraction: distill and describe essential properties, while ignoring (hiding) some details

- I ignore details only for now, I will need bring them to the picture later

FRS requirements



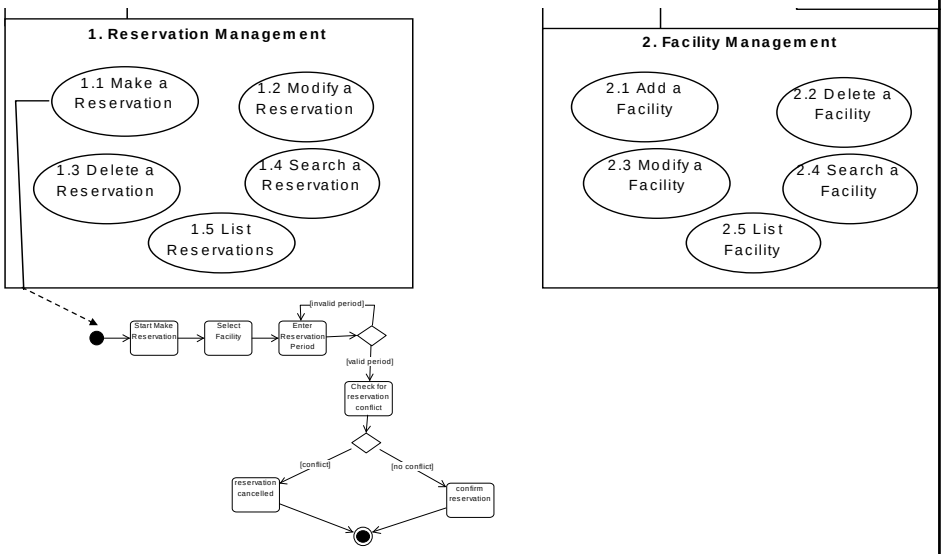
- Still, there is huge gap between the left and the right!

How can I fill this gap? More abstractions?

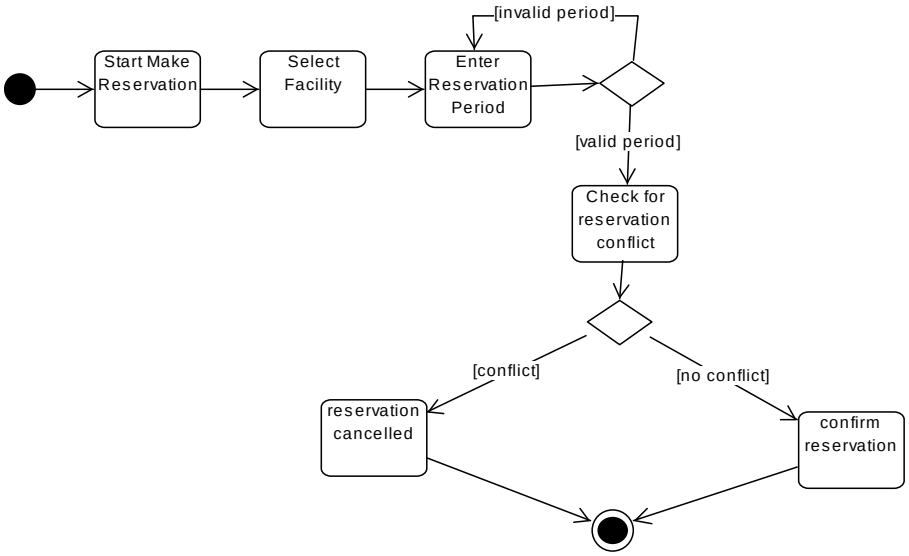
Models!

- We have UML – Unified Modeling Language
- Models are pictures and "one picture is worth thousands words"
- Understand software through pictures!
- Models allow us to look at software from many angles – structure, behavior, …
- Each model reveals some focused, abstract system view

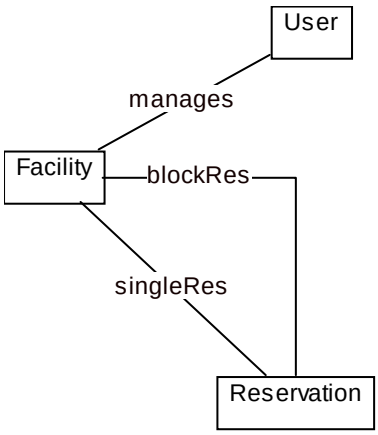
Use Case diagram



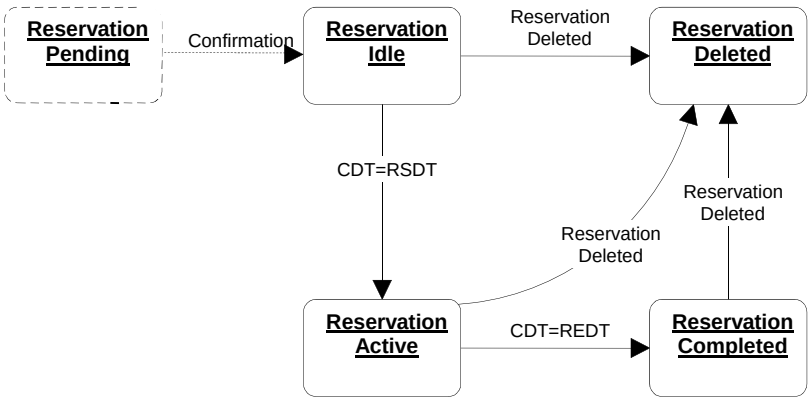
Activity diagram: *Make Reservation*



Conceptual Class diagram



State Transition diagram



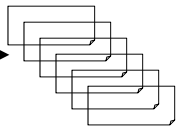
Abstractions help

FRS requirements

Models

Documentation

FRS code



- The gap between the left and the right is still wide!

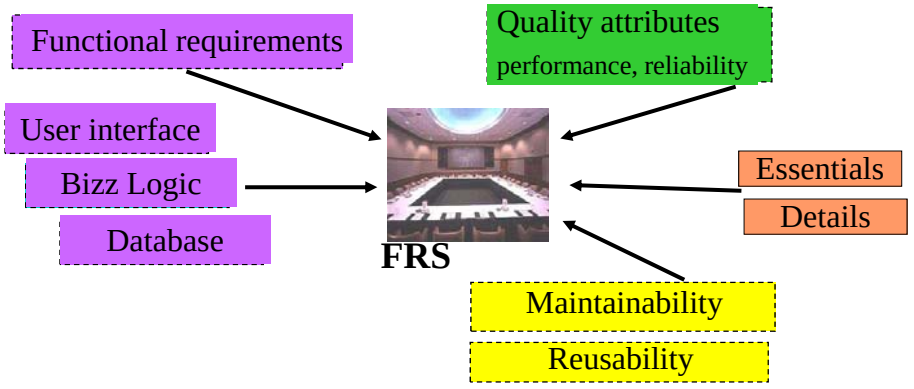
What else can we do to fill the gap and to further conquer complexity?

When we develop software, we are concerned with so many things:

- FRS function (What?) and its design (How?)
- FRS user interface, business logic, database
- Quality attributes:
  - Usability: Is FRS easy to use?
  - Performance: is FRS response quick enough?
  - Reliability: won't it fail when I need it most?
- Maintainability: Is FRS easy to change?
- Reusability: Can I reuse FRS components?

Can I deal with each concern separately from others?

- Separation of Concerns is a common sense approach to manage complexity



2nd principle: Separation of Concerns (SoC)

The power of SoC:

- Understand one concern separately from other concerns
- Work on concerns separately
  - Important in team projects

Some concerns are easier to separate than others:

- Most often I can separate user interface from business logic

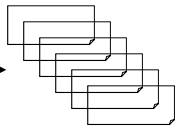
But

- It is difficult to separate performance concern from other aspects of a software system (i.e., other concerns)

Abstraction and SoC

Abstraction

Separation of Concerns



FRS requirements

FRS code

- The gap between the left and the right is still wide!

What else can we do to fill the gap and to conquer complexity?

Calculation example

- What is 2556 * 1551 = ?

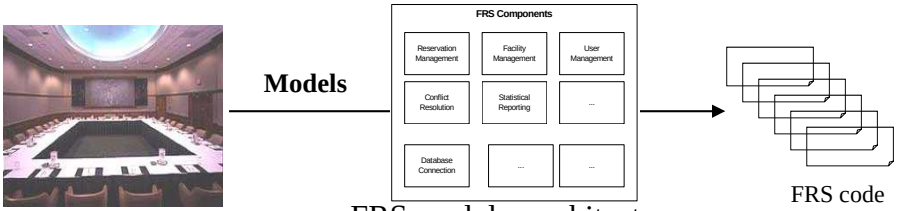
```
   2556
  12780
12780
2556
-------
3964356
```

What principle is at work here?

- Decomposition: solve a complex problem in steps:
  - decompose it into parts, understand and solve each part
- "Divide and Conquer" tactic – who used it first?

3rd principle: Modularity, Decomposition

- We decompose along two dimensions:
  - *Product*: decompose system into parts
  - *Time*: Decompose process into steps
- Each dimension has its own complexities, so we deal with them separately (which principle?)

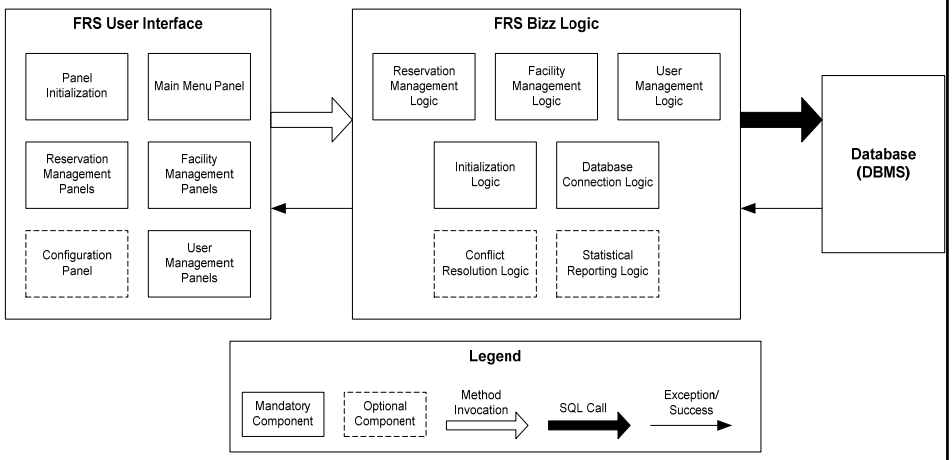


FRS requirements

FRS modular architecture
Macro-structure

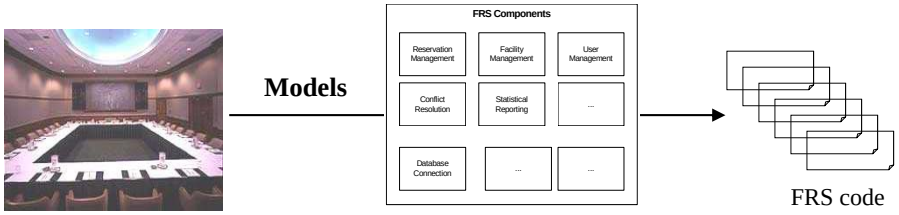
FRS code
Micro-structure

FRS modular architecture



boxes are units of decomposition: Modules (or Components)

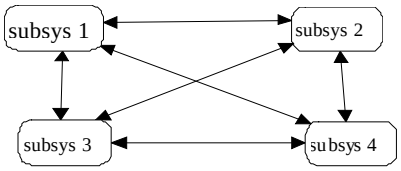
How architecture helps?



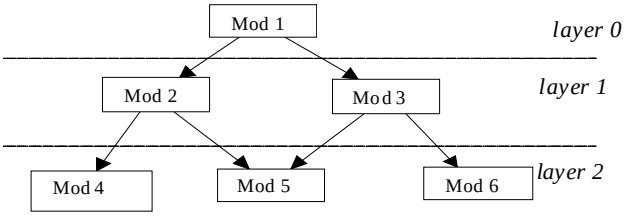
- Understand code at the higher abstraction level
  - architecture shows logical groupings of modules that play different roles
- Explain how FRS requirements have been implemented
- Evaluate and justify design decisions

Typical levels of decomposition

Decomposition of a system into subsystems:



Decomposition of a subsystem into a layered hierarchy of modules:



How to decompose?

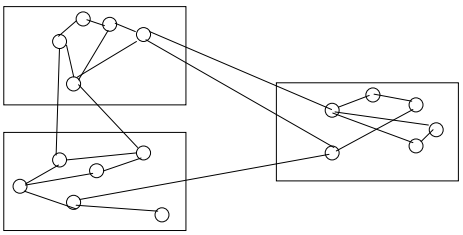
- It is a key to the success
  - Most experienced developers in a company do that

1. Make modules highly cohesive
   - Focused on a group of highly related tasks
   - An easy test: describe the modules role and function
2. Minimize couplings among modules
   - Any interactions or dependencies
3. Organize modules into layers
   - Strive for simple interactions among layers

Coupling and Cohesion

- We want this:

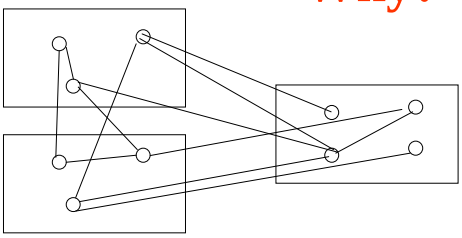
high cohesion and
low coupling



Why?

- But not that:

low cohesion and
high coupling

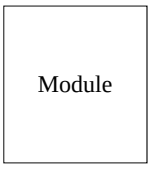


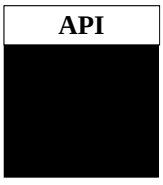
High cohesion and low coupling

- High cohesion and low coupling is a good design - why?
  - we can treat modules as black boxes, independently of each other
- But why is it good to have black-box modules?
  - We can work on modules independently
  - We can change one module with minimum impact on other modules (ripple effects of changes)
- The above are good enough reasons to strive for high cohesion and low coupling modules

Can we achieve total module independence?
Zero module coupling?

Module interfaces - APIs





Public: interface

Hidden, Private: implementation

- We want to let others use our module without knowing all the module details - how?
- We want to change a module without affecting other modules – how?
- API: a description of what you can do with a module

Abstraction and information hiding

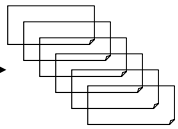
Principles so far

FRS requirements

Abstraction

Separation of Concerns

Modularity



FRS code

- The gap between the left and the right become narrower!

What else can we do to fill the gap?

To conquer complexity?

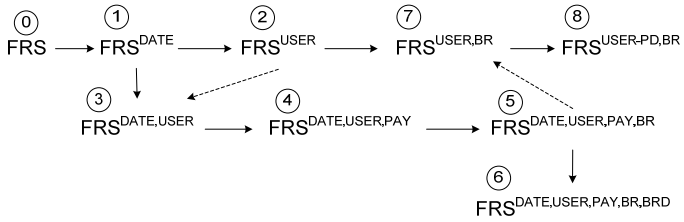
4th principle: Generality

- Why to solve just one problem if I can solve many similar problems with one program?
- Examples:
  - Generic <T> Stack
    - Type parameter T: IntStack, ShortStack, CharStack
  - GUI for user interface generation
  - Compiler-compiler:
    - A single system that generates compliers for many programming languages
    - A general solution to building compilers
- Possible trade-ff: efficiency
  - Specialized solution is often more efficient than general one

5th principle: Design for Change

Why is it a principle?

- Change is inevitable, we better plan for it
- Iterative software development won't work if software is hard to change
  - Agile methods, daily/weekly builds
- FRS evolution: multiple versions released to customers



6th principle: Rigor & Formality

- We all love free, happy creativity!
  - so why spoil the fun of programming with rigor?

A. You experiment with a new idea, sketch initial design, try a new tool – do you need rigor? **No**

B. You already reviewed and accepted design decisions and now document it - do you need rigor? **Yes**

How about programming itself?

3. Program code is a 100% formal object

- Still, at times formality can help!

Comments on being formal

- Elements of formal notations can help with APIs. however:
  - Give priority to intuitive descriptions
  - Use formal notations to make your description precise
  - Embed formal descriptions in intuitive, informal descriptions that are easy to understand
- Always need choose the right technique for the task in hand
  - The right formal notation for the purpose
  - The right model for the purpose
  - The right abstraction for the purpose

That's all about principles

Hope you will find some of this useful in your projects!

--- The End ---