SE3430/5430 OOAD F14
Lecture Notes 01

Kun Tian

What is needed for AD to build a good system?

- Lecture Notes:
- Using UML, Stevens: Chapter 1 and Chapter 2.
- UML Distilled, Fowler: Chapter 1
- Note: Both books are based on UML 1.4. UML 2.x is the current release.

What is a good system?

- **Describe a good system, what do we want it to be?**
- **Useful / Features** – does something that provides a benefit
- **Usable** – is easy to use, doesn’t required an absurd effort to derive benefit.
  - Does anyone have experience on a software that was awkward to use?
- **Reliable** – produces desired output with few failures
- **Flexible/ Extensibility** – does many useful things (not just add a fixed list of numbers), can be fixed if defects found, and can be extended for new functions.
- **Affordable** – the economics of the system are positive.
- **Performance** – does it produce results in a reasonable time.
  - Some results are time sensitive, for instance an aircraft collision avoidance system must be able to figure out that a collision is imminent while there is still time to avoid the accident.

What is a good system?

- **Scalability** – can be enhanced to handle additional capacity (e-commerce sites)
  - Still functions well the requests for service skyrocket ☺
- **Availability** – can be readily used
  - “Always” be there
- **Compatibility?**
  - One works with another?
- **Correctness**?
  - What if the calculator program gives wrong outputs?
- **Reusability?**
  - Reusable is good, why?
- **Maintainability?**
  - Why important? Majority of software cost goes into maintenance.

Do we always have a good system?

- Problems that happen along the way
  - Capturing the users' current needs – valid requirements
  - Keeping track of users' needs as they change – challenge is SE and RE
  - Estimating efforts – if we are not diligent in our planning, things are generally more complex and take longer than we initially estimate - challenge in Project Planning
  - Keeping everyone on the team working productively – specialists / non-specialists – challenge in Project Management
  - Quality, a big issue! What is the fundamental objective of SE?

Do we have a good system?

- Problems that happen along the way
  - Support – training, user documentation, supporting multiple older versions of software.
  - Technology may change – look back to games 15 years ago, what are the differences?
  - 3rd party dependencies
- What happens if we fail in the above categories?
  - Some or many characteristics of a good system will be missing
  - Examples of failures.(Correctness) Y2k, Mars Lander(s), Patriot Missile Problems, etc.

Why is it difficult for us to have a good system? Why do we need AD for SE?

- Fundamental Problem: There is a limit to how much a human can understand at any one time.
  - "Heroic Programming" only works for small projects. It doesn't apply to the current day software projects anymore. Why? Increasing sizable and complicated projects.
  - It is impossible for a developer or a maintainer to understand everything about the system all at once.

Why is it difficult for us to have a good system? Why do we need AD for SE?

- Untangle the Intricacy of software.
  - Consider a maintainer trying to make some changes to a system. He changes 3 lines of code, and what else does he/she need to do?
    - He/she may need to make corresponding changes to the parts of the system that will be affected by the new 3 lines of code. Since he/she cannot understand everything about the system at any one time, he/she may fail to make all the necessary changes to the system. Then he/she may risk introducing bugs. The system is not good any more.
    - Example, Angry Birds (size, mass, behavior -> physics -> graphics -> almost everything)
  - Here, the challenge is program reasoning, "How do we identify which parts of a system need to be changed as a result of a specific change made to the system in order for the system to be still GOOD?" The Dependency Problem.

The Dependency Problem

- Dependence – Module A depends on Module B: B changes -> A changes.
  - Is this good? Why? Many dependencies lead to what? Disaster in Program Reason.
  - Goal: we try to minimize dependencies, or in other words to achieve minimal coupling and high cohesion.
    - Low coupling = low dependencies between modules =(almost) high cohesion = put highly coupled features into a single module as much as possible.
    - High coupling = high dependencies between modules = (almost) low cohesion = put highly coupled features into different modules as much as possible.

Structures/Modules

- Solution: Injecting structures/modules into software to improve reasoning.
- Structure/Modules – somewhat generic term for a chunk of a system that makes sense for it to be considered separately. (Files, subroutines, libraries, classes, etc.)
  - Firstly proposed in the 1970s by a paper addressing Separation of Concerns. What is it (Concern)? What is SOC? Single-mindedness.
  - SOP =Modularization: We modularize a system into modules to achieve high modularity.
    - Rule of Thumb: A good system should be modularized.
  - Then we resolve the Dependency Problem by identifying and minimizing dependencies between Modules.

Roadmap

- A: There is a limit as to how much a human can understand at any one time.
- B: People usually cannot understand all of the system by themselves. (Even for a team that is still difficult)
- C: If we introduce some changes to some modules, we may need to change the other modules of the system to accommodate the changes in order for your system to be still **GOOD**.
- D: Our system may have high dependencies between modules.
- E: We(or our team) may fail to make some necessary changes to some modules, and as a result your system becomes **BAD**.
- F: One way to make sure that we can still produce good system is to cope with our inability to understand all parts of a system by improving program reasoning and minimizing dependencies between modules (high cohesion and low coupling).
    - A=>B;
    - B, C, D=> E =>F;

Interfaces

- A technology that supports our goal.
- Every module has an interface. Put contracts of modules onto their interfaces. Abstract the services into interfaces.(Abstraction)
- The clients can only use the features of a server module through an interface. An interface hides the details of providing the service from the clients – it encapsulates the details (Encapsulation).
- Limiting dependency: If we successfully document all the assumptions in the interface we will be able to say: If a module changes internally without changing its interface, this change will not necessitate any changes anywhere else in the system. (Contract remains the same.)
- A useful way of thinking about Modules is by classifying them as service providers (servers) or service users (clients). Module A uses Module B's service, Module A is a Client, Module B is a Server

Interfaces

- Helps Encapsulation (aka, Information Hiding) – prevent access to the details of how a module provides its service. A module provides no more than enough information than its clients requires. (PRIVATE functions and data, accessors.)
- Example, store interface – we understand the various interfaces for a pizza store. What you care about it is how to order a pizza.
  - Do we need to understand the details of making a good crust or sauce to order a pizza?

Interfaces

- Encapsulation (interface) helps: Example, the Y2K problem: Which would be easier to fix?
  - a) manipulation of dates all over a system (for example: couts, i/o, db transactions, manipulations, checks, etc.),
  - or b) a single class, let's call it Date, that contains all of the above functionality and provides all date services to the rest of a system? (put all the date manipulation functions into a single module)
  - So it helps achieve low coupling and high cohesion

Interfaces

- Every interface is associated with some assumptions.
- Checking the Assumptions of Interface (find out the dependencies)
  - **Syntactic dependency checking** – check the parts of an interface for type and number consistency; like is done by a compiler. Strongly typed languages do this at compiler time and prevent us from messing up type sensitive information.

- float paid = 15.0, cost = 10.0;
- Int refundIndex = paid / cost;
- if(refundIndex > 1) refund( );
- //A very strongly typed language would warn you of a possible loss in precision. (we need check it semantically)

Interfaces

- Checking the Assumptions of Interface (find out the dependencies)
  - **Semantic dependency checking** – go beyond syntax, to test whether the services provided by a module are consistent in meaning with the use of the client.
    - Server was giving range as floating point numbers, Client was using these numbers as a range, there was not a Syntactic dependency problem (floats), but there was a semantic mismatch in that one module thought the numbers should be in English units and the other thought they were metric units.

Modularity

- In software design, modularity refers to a logical partitioning of the "software design" that allows complex software to be manageable for the purpose of implementation and maintenance.
  - Divide and conquer.
  - Anything that reduces what we need to know is helpful when we are dealing with large systems (for example a system with 10,000,000 SLOC).
  - We do **not** need to know the details of how all servers work, just how to use their interfaces.
- Bugs (actually we should refer to these as defects) should be easier to isolate if specific functionality is isolated/modularized (to increase cohesion as well). How?
- It makes reuse possible. How?

Modularity

- A module may have multiple interfaces.
  - Sometimes it is convenient to document the services that a module provides as several different interfaces (abstracting functionalities into interfaces), so that we can be more precise about what services a given client needs.
    - Warning: but too many interfaces usually means a module may not be cohesive or coupling may be high.
  - Rule of Thumb: A good system consists of abstracted (encapsulated) modules, which in turn have one or multiple but not too many interfaces.

Cohesion, Coupling, Encapsulation, Abstraction

- Cohesion (of a module): how well the services that it provides are related to a single well-defined theme/purpose?
- Coupling (between modules): how well different modules are coupled together?
- Difference between Encapsulation and Abstraction is Encapsulation prevents clients from seeing the details, whereas Abstraction reduces the amount of information that a client needs to know to obtain a service.

Software Architecture

- Bring the program reasoning problem to the system level.
- Architecture and Components
- Component – some chunk of software that can be reused, or can say it's a module of your system.
- Software Architecture: The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.
- Software Architecture defines the relationships between components (modules) as relations, so we can get to know which modules are the clients for which modules.
  - Rule of Thumb: A good system consists of a clear architecture.

How are good systems built?

- It's NOT just a random process!
- We DON"T just start coding!
- Use a defined process
- Have a clear set of requirements to satisfy
- Regard verification and validation as important
- Keeps project knowledge, architecture and components under configuration control
- Learn from experience – if something in the process doesn't work, we fix it so that it will not be a problem next time. Continuous Process Improvement.
- Makes use of appropriate methodology for all of the above task (OOAD)