Software engineering

> multi-person construction of multi-version software

Author: Daria Shutina

- Software engineering
  - 23-02-02
    - Organization stuff
    - Failures and Catastrophes
    - How to Program?
  - 23-02-07
    - Socio-technical systems
    - Emergent Properties
    - Requirements Engineering
      - Schematic RE Procedure
  - 23-02-21
    - UML
      - Diagram Types
        - Use Case Diagrams
          - Example
        - Activity Diagrams
          - Example
        - Sequence Diagrams
          - Example
        - State Transition Diagrams
        - Class Diagrams
    - DSL
  - 23-02-28
    - Design patterns
      - Types of patterns
      - Singleton pattern
        - Example: creating a singleton class
      - The Observer Pattern
        - Example
      - The Mediator Pattern
      - The Facade Pattern
      - The Proxy Pattern
        - Example
      - The Adapter Pattern
      - Composite Pattern
  - 23-03-14
    - Compile/Link Steps
    - The C Preprocessor
    - The C(++) Compiler
    - Object Files
    - Name mangling

23-02-02

Organization stuff

Course link: https://peter-baumann.org/Courses/SoftwareEngineering/index.php

Timetable:

- Tuesdays 8:15 -- helpdesk online
- Tuesdays 9:45 -- lectures offline
- Thursdays 11:15 -- helpdesk online

Project:

A group of 4-5 people submit a design doc, then there are 2-week phases:

- Specification & design (teams of 5 – free to join up)
- Code sprints Implementation (teams of 2 – random)

Grading:

Software Engineering *lecture* (2.5 CP) = 33%

Software Engineering *project* (5 CP)   = 67%

Failures and Catastrophes

http://catless.ncl.ac.uk/Risks/

Lessons from cases:

1. ***don't optimize to death.***

   Y2K: "1997" in COBOL stored as "97", and "2000" -- as "00".
2. ***be careful with guessing user intent. Users need guidance.***

   a bug in FORTRAN code: "DO 20 I = 1,100" and "DO 20 I = 1.100"

How to Program?

1. Write code
2. Test it on a few samples
3. Bug fixing, improving efficiency
4. GOTO 1

This is usually appropriate for 1-person projects. In real-life projects there are different people $\Rightarrow$ different ideas through years. Also, if developer $\neq$ user, there is a frequent dissent about expected vs. implemented functionality.

Common problems for SE projects are mostly about organization:

- Complexity of the idea
- Communication (b2b, b2c)
- Flexibility: change of requirements, components, methods, tools over lifetime
- Lack of education
- Bad project management

23-02-07

Socio-technical systems

Socio-technical systems are hardware & software + operational processes + people. It applies an understanding of the social structures, roles and rights to design systems that involve communities of people and technology. Examples of STSs include emails, blogs, and social media sites such as Facebook and Twitter.

STSs seek to merge people and technology, viewing the integration of computers into societal systems. It results in designing a complex system for the product, and properties of the system depend on components and their relationships. The computer will always do something, but it will do it correctly only in case it has been programmed for that situation.

System's behavior partially dependent on human operators and a time-varying environment. It does not always produce same output when presented with same input.

If you do not understand the organizational environment where a system is used, the system is less likely to meet the real needs of the business and its users.

Emergent Properties

Emergent Properties are the result of various system components working together, they do not belong to an individual component.

- Volume: total space occupied depends on how component assemblies are arranged & connected.
- Reliability: unexpected interactions can cause new types of failure.
- Security: attacks not anticipated by system designers may defeat built-in safeguards.
- Repairability: in order to fix the problem, can the system be rebooted or turned off while being repaired?
- Usability: it is about how easy it is to use the system (example: presenting info in different languages).

Requirements Engineering

Requirements engineering is the process of defining, documenting, and maintaining requirements in the engineering design process. It is the process of eliciting the services that the customer requires from a system and the constraints under which it operates and is developed

Types of Requirements:

- User requirements: statements in natural language and diagrams of the system’s services and its operational constraints.
- System requirements: a structured document with detailed descriptions of the system’s functions, services, operational constraints and technical effects.

Schematic RE Procedure

1. Inception — ask questions
    - basic problem understanding
    - Identify stakeholders
    - Establish trustful communication
2. Elaboration — create analysis mode
3. Negotiation — determine each stakeholder's "win conditions", negotiate "win-win"
4. Requirements management — a set of techniques for documenting, analyzing, prioritizing, and agreeing on requirements
    - review mechanism (errors, missing info, feedback)
    - document versioning

23-02-21

UML

The UML -- Unified Modeling Language -- is the standard language for specifying, visualizing, constructing, and documenting all the artifacts of a software system. Synthesis of notations were done by Grady Booch, Jim Rumbaugh, Ivar Jacobson, and many others.

Diagram Types

The purpose of diagrams can be conceptual, for specification and for implementation.

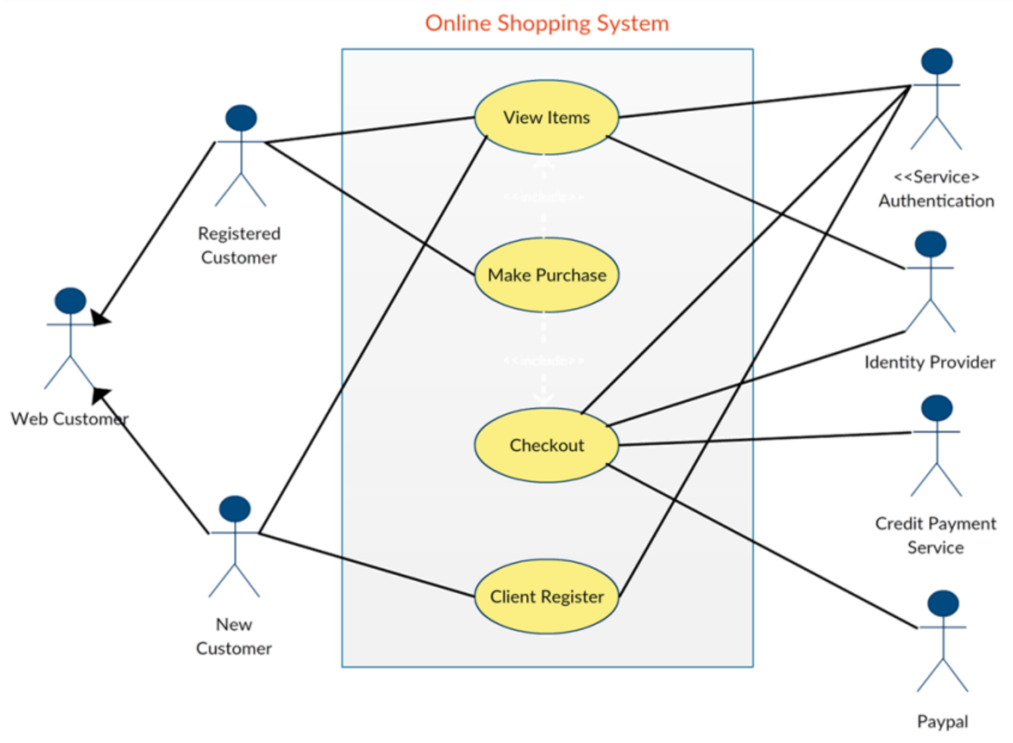
Use Case Diagrams

- use case = chunk of functionality, not a software module. Should contain a verb in its name
- actor = someone or something interacting with system under development. Like a role in a scenario

  Relationships between actors and use cases are visualized.

Example

People are actors, circles are use cases.

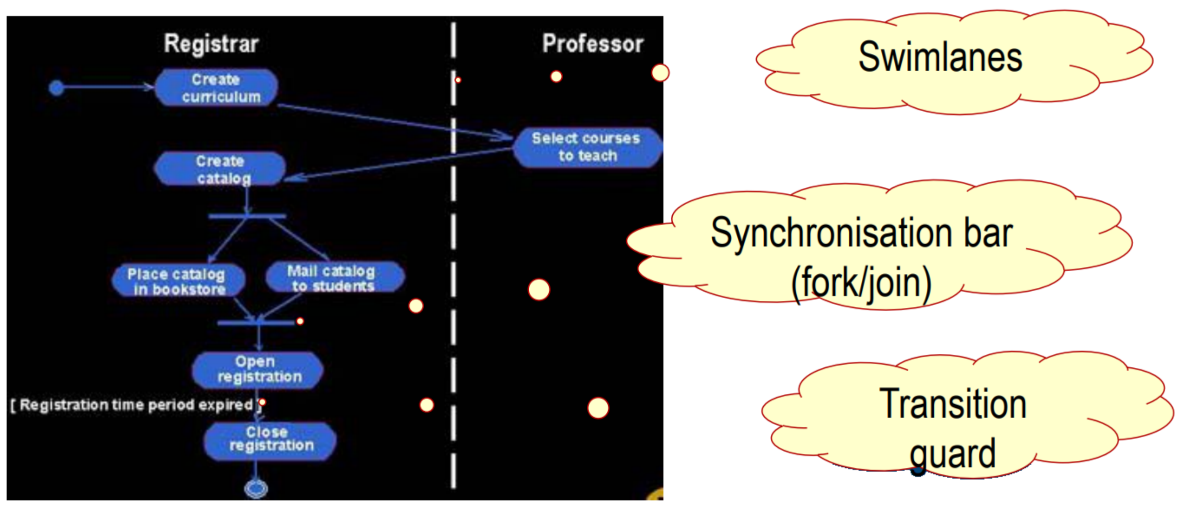


Activity Diagrams

Graphical workflow of activities and actions.

There is a start point, then ways of doing something synchronously.

Example



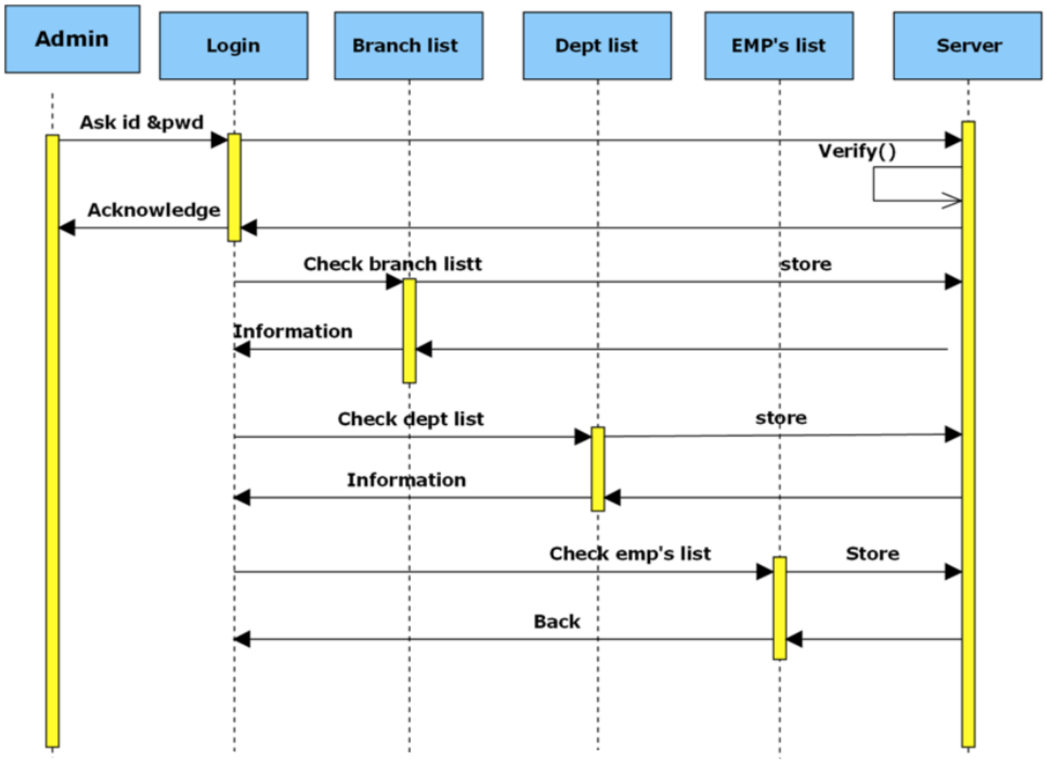
Sequence Diagrams

Displays object interactions (a sequence of moves) arranged in a time sequence.

A SD should be done for every use case (for every actor of the scenario).

Example

Yellow bars are a grouping of actions. For example, `ask id & pwd` and `acknoledge` are actions that are done together.



State Transition Diagrams

Shows life history of a given class.

STDs are usually used for classes that typically have a lot of dynamic behavior.

There are a start point and finish point(s). Any constraints are also shown on the diagram.

Class Diagrams

Class = collection of objects with common structure, common behavior, common relationships, and common semantics

Properties of class diagrams:

- Association ($f(a, b) = f(b, a)$)
- Aggregation
- Dependency
- Inheritance

DSL

domain-specific modelling languages. Were created, since UML considered (too) complex (general-purpose) and software biased. For example, SQL is a DSL.

In whole, UML is better for enterprise apps (millions of possible directions) and DSL is better for embedded systems (clearly delimited app domain & paths).

23-02-28

Design patterns

Types of patterns

- creational

  • Abstract Factory -- Creates an instance of several families of classes

  • Builder -- Separates object construction from its representation

  • Factory -- Method Creates an instance of several derived classes

  • Prototype -- A fully initialized instance to be copied or cloned

  • Singleton -- A class of which only a single instance can exist
- structural

  • Adapter -- Match interfaces of different classes

- • Bridge -- Separates an object’s interface from its implementation
- • Composite -- A tree structure of simple and composite objects
- • Decorator -- Add responsibilities to objects dynamically
- • Facade -- A single class that represents an entire subsystem
- • Flyweight -- A fine-grained instance used for efficient sharing
- • Proxy -- An object representing another object

- behavioral (mediator, strategy patterns)
  - • Chain of Respect -- A way of passing a request between a chain of objects
  - • Command -- Encapsulate a command request as an object
  - • Interpreter -- A way to include language elements in a program
  - • Iterator -- Sequentially access the elements of a collection
  - • Mediator -- Defines simplified communication between classes
  - • Memento -- Capture and restore an object's internal state
  - • Observer -- A way of notifying change to a number of classes
  - • State -- Alter an object's behavior when its state changes
  - • Strategy -- Encapsulates an algorithm inside a class
  - • Template Method -- Defer the exact steps of an algorithm to a subclass
  - • Visitor -- Defines a new operation to a class without change

Singleton pattern

It is about creating only one instance of a class that is shared between other objects in the application.

The main downside of this pattern is the possibility of race conditions, when several objects try to write into the same variable.

Example: creating a singleton class

```
// fancyLogger.js

class FancyLogger {
    constructor() {
        if (FancyLogger.instance == null) { // <=> no instances yet
            this.logs = []
            FancyLogger.instance = this
        }
    }

    log(message) {
        this.logs.push('${message}')
    }

    printLogCount() {
        console.log('${this.logs.length} logs')
    }
}

const logger = new FancyLogger()
Objects.freeze(logger) // now FancyLogger class cannot have new methods or variables

export default logger
// if there is `import` in another file, we will export the instance, not the class
```

```
// simpleFile.js

import logger from './fancyLogger.js'

export default function simpleFunction() {
    logger.log('aboba')
    logger.printLogCount()
}
```

The Observer Pattern

The Observer Pattern defines a "one-to-many" dependency between objects (there is one object A and several objects which are dependent on A). When one object changes state, all of its dependencies are notified and updated automatically.

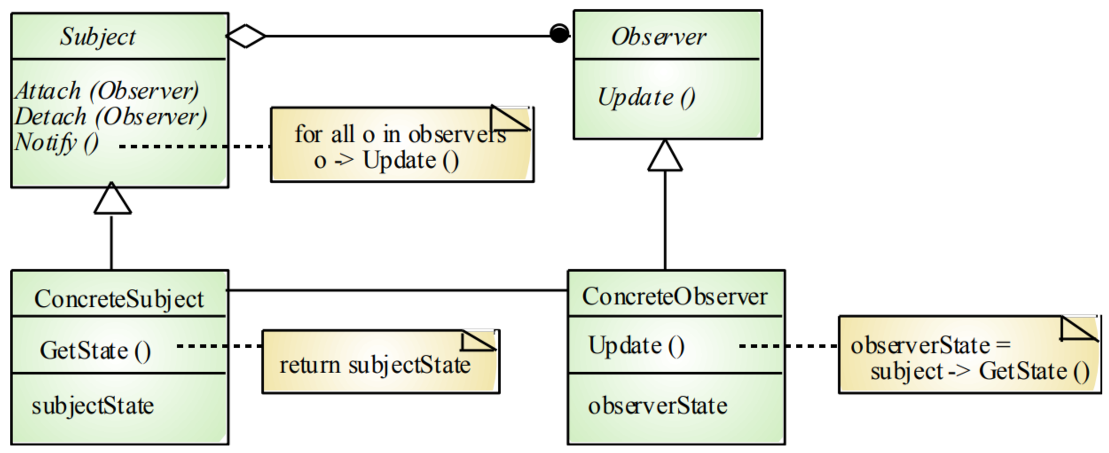
Object A above is called *subject*.

Objects dependent on A are called *observers*.

Relations between the subject and dependencies are called *subscriptions*.

Why The Observer Pattern is important? Imagine we have several changes in the timeline. And every observer will check periodically (by itself) whether something has changed recently. Thus, we get wasted cycles that are multiplied by the number of observers. This is a bad pattern for the observer being responsible for retrieving data.

Example



The Mediator Pattern

The Mediator Pattern defines an object that encapsulates how a set of other objects interact with one another. In other word, the mediator controls the communication between objects.

As opposed to the Observer Pattern, objects send request to the mediator and get the answer.

The Facade Pattern

We need a centralized place to put all of the logic inside it and to restrict direct access to the inner workings of the library/framework/other complex class.

The Facade Pattern provides a simplified interface to a set of interfaces in a subsystem. This interface has methods with access to a particular part of the subsystem's functionality.

The main idea of the pattern is to make the subsystem easier to use. Instead of calling several functions, you have a method with a reasonable name, and a piece of logic is stored inside this method.

The Proxy Pattern

To begin with, the proxy-server acts like a firewall, a filter and a caching tool. It provides a high level of security and data protection by checking the data it receives.

The Proxy Pattern provides a substitute (a placeholder) for the original object. The substitute controls access to the original object by performing something before or after the request reaches that object.

Example

We have a login form on the site. Before the user will see the personal account, he/she should enter correct login credentials. The proxy checks checks entered credentials and allows or denies the access.

The Adapter Pattern

It is about making two incompatible interfaces compatible.

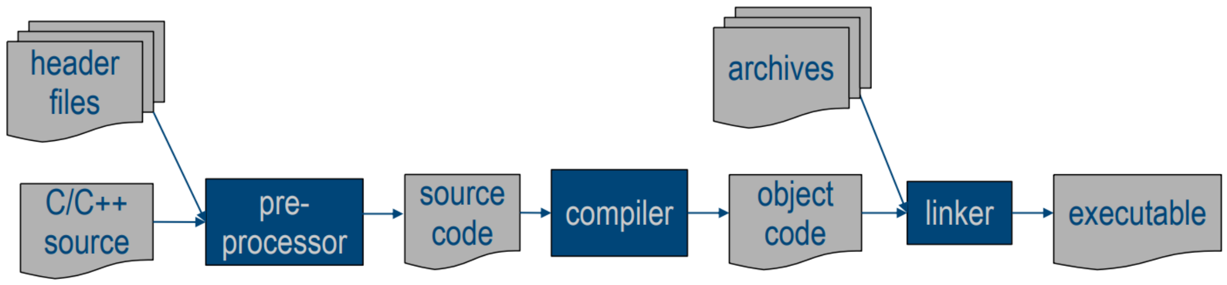
Composite Pattern

It is about composing a set of objects into a tree and then work with trees as if they were individual objects.

All elements share a common interface allowing the client to treat individual objects and compositions (subtrees) in the same way.

23-03-14

Compile/Link Steps



The C Preprocessor

Things with which the preprocessor works:

- defines commonly used constants, code fragments, etc.
- include guards in header files
- include headers (`#include <stdio.h>`)
- process macros. If you use a function as a macro, do not forget to put brackets around arguments:

```
// #define mult(a,b) a*b  <--- bad for a=2+3, b=4
#define mult(a,b) ((a)*(b))
```

In Linux, header files live in `/usr/include` and `/usr/local/include`.

The C(++) Compiler

Generates relocatable machine -- „object“ -- code from source code. The code can be saved in different memory segments.

Object Files

Contain machine code of source file. There is info about constants, size of static data segments, etc.

`objdump` command displays information from an object file.

Name mangling

Compiler modifies a function name to make it unique.

Every compiler has its individual mangling algorithm. Thus, code compiled with different compiles is incompatible.

C language does not know about name mangling. You need to use `extern "C" {}` keyword in order to avoid name mangling.