Software engineering

> multi-person construction of multi-version software

Author: Daria Shutina

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.

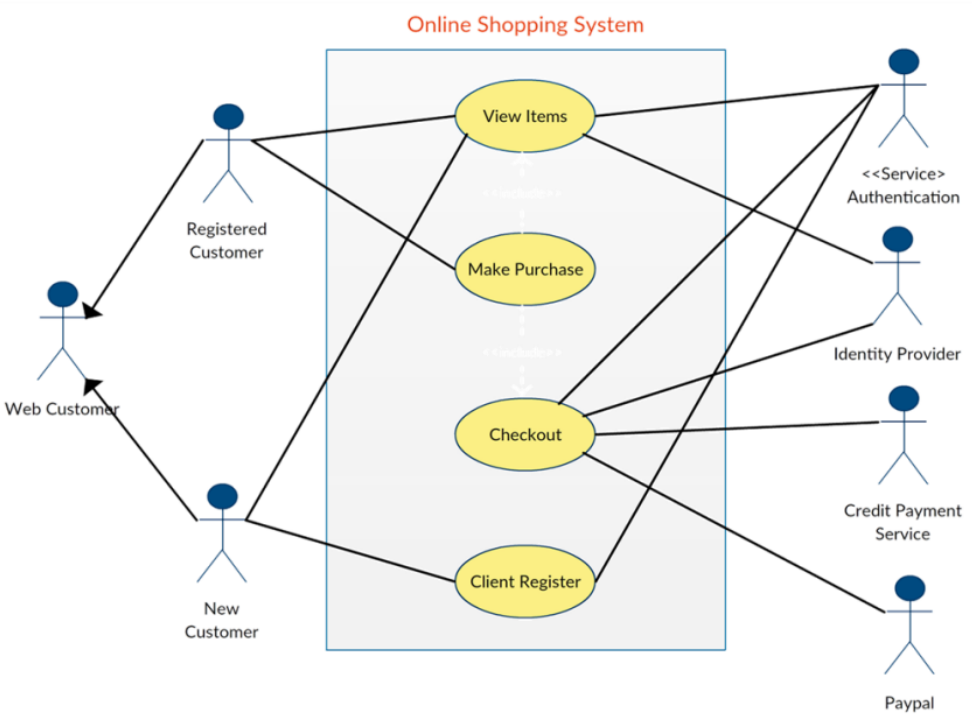
Use Case Diagrams

- 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.

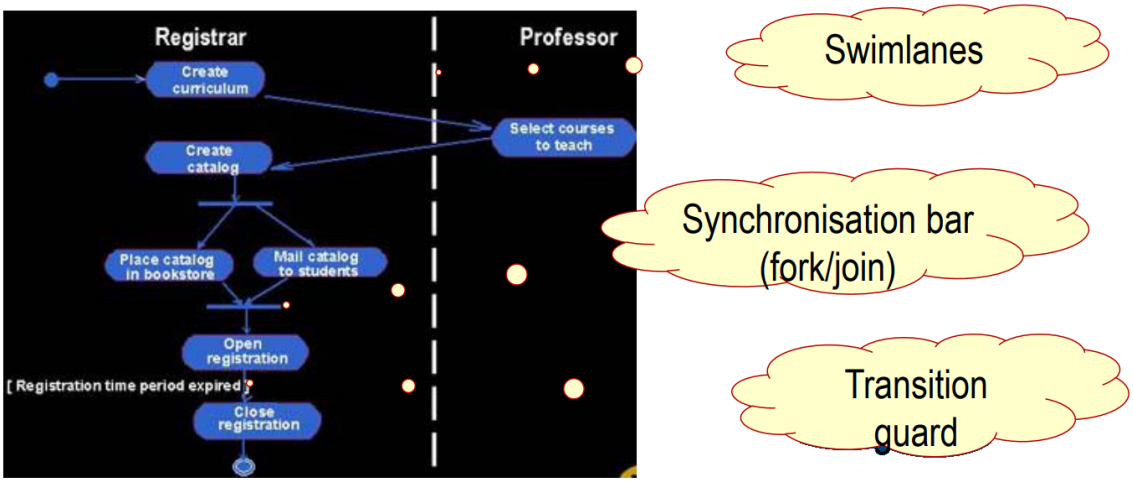


Activity Diagrams

Graphical workflow of activities and actions.

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

Example



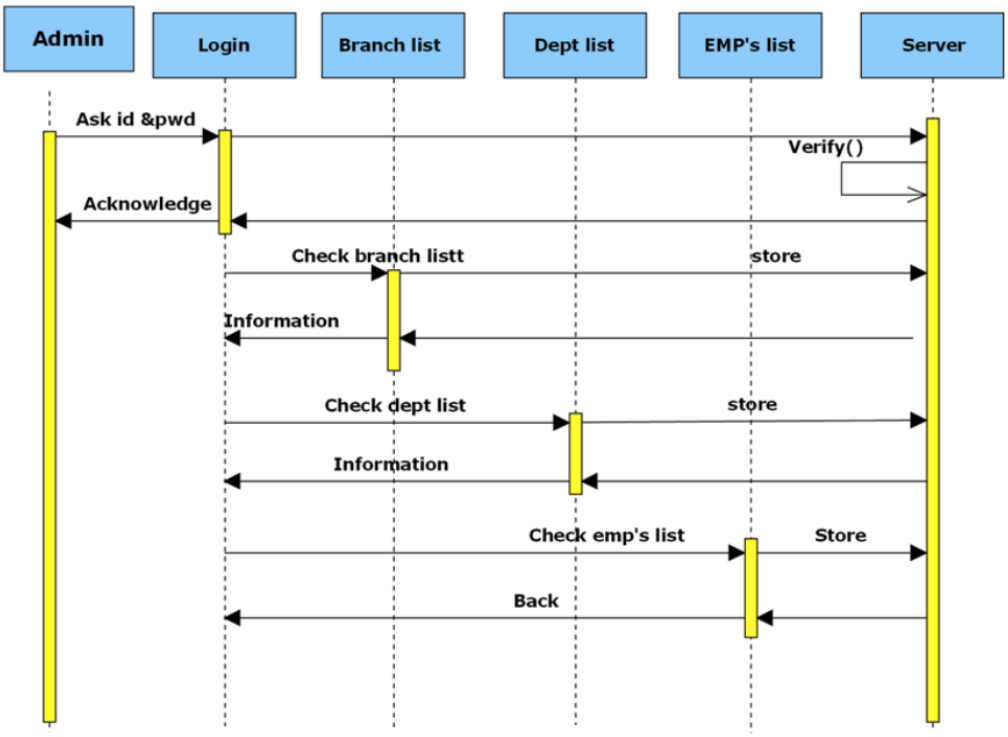
Sequence Diagrams

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

Software architecture

Software architecture -- SA -- a pattern, which describes all inner components of the system and interactions between them. You can visualize SA through UMLs.

For a better understanding, there are a few basic types of software architecture:

- **Application architecture** -- focuses on the structure and organization of the software app. It considers aspects like the choice of programming languages, frameworks, libraries, and patterns to achieve the desired functionality and maintainability. Example: monolithic, microservices, and layered architectures.
- **Information architecture** -- deals with the structure and organization of information within a system or a website. It focuses on how information is categorized, organized, and presented to users. It is about creating site maps, navigation menus, content categorization, and defining metadata.
- **Database architecture** -- refers to the design, development, implementation and maintenance of a database system. There are various database architecture models, such as hierarchical, network, relational, and object-oriented.
- **Network architecture** -- involves the design and organization of computer networks. It defines how devices are connected and communicate with each other. Network architecture includes decisions about network topology, protocols, addressing, security measures, and network hardware. Examples: client-server, peer-to-peer, and hybrid architectures.

Design patterns

Creational patterns

Provide more flexibility in how the objects are actually created.

Singleton

Guarantees that only one instance of the class exists.

Simple implementation:

```
// `instance` field is static, since `getInstance` method needs to be static
public class Singleton {
    private static Singleton instance;
    private String data;

    private Singleton(String data) {
        this.data = data;
    }

    public static Singleton getInstance(String data) {
        if (instance == null) {
            instance = new Singleton(data);
        }
        return instance;
    }
}


class Main {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance("aboba");
    }
}
```

Thread-safe implementation:

```
public class Singleton {
    private static volatile Singleton instance;
    private String data;

    private Singleton(String data) {
        this.data = data;
    }

    public static Singleton getInstance(String data) {
```

```
        // use `result` variable, since `instance` is volatile
        // thus, we read directly from the main memory only once
        Singleton result = instance;
        if (result == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton(data);
                }
            }
        }
        return result;
    }
}
```

Factory

Separates the product's construction code from the code that uses this product.

It relies heavily on inheritance.

```
// Product interface
interface Product {
    void use();
}


// Concrete products
class ConcreteProductA implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA");
    }
}

class ConcreteProductB implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductB");
    }
}


// "Factory" class
class Creator {
    public static Product createProduct(String type) {
        if (type.equalsIgnoreCase("A")) {
            return new ConcreteProductA();
        } else if (type.equalsIgnoreCase("B")) {
            return new ConcreteProductB();
        }
        throw new IllegalArgumentException("Invalid product type: " + type);
```

```
    }
}


// the products are used here
public class Main {
    public static void main(String[] args) {
        Product productA = ProductFactory.createProduct("A");
        productA.use();  // Output: Using ConcreteProductA

        Product productB = ProductFactory.createProduct("B");
        productB.use();  // Output: Using ConcreteProductB
    }
}
```

Abstract factory

Allows to produce a group of related objects without specifying their concrete classes.

```
// Abstract Product A
interface ProductA {
    void use();
}

// Concrete Product A1
class ConcreteProductA1 implements ProductA {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA1");
    }
}

// Concrete Product A2
class ConcreteProductA2 implements ProductA {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA2");
    }
}




// Abstract Product B
interface ProductB {
    void interact(ProductA productA);
}

// Concrete Product B1
```

```
class ConcreteProductB1 implements ProductB {
    @Override
    public void interact(ProductA productA) {
        System.out.println("Interacting with ConcreteProductA1");
        productA.use();
    }
}

// Concrete Product B2
class ConcreteProductB2 implements ProductB {
    @Override
    public void interact(ProductA productA) {
        System.out.println("Interacting with ConcreteProductA2");
        productA.use();
    }
}









// Abstract Factory
interface AbstractFactory {
    ProductA createProductA();
    ProductB createProductB();
}

// Concrete Factory 1
class ConcreteFactory1 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA1();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB1();
    }
}

// Concrete Factory 2
class ConcreteFactory2 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA2();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB2();
    }
}
```

```
88
89  // Client code
90  public class Main {
91      public static void main(String[] args) {
92          AbstractFactory factory1 = new ConcreteFactory1();
93          ProductA productA1 = factory1.createProductA();
94          ProductB productB1 = factory1.createProductB();
95          productB1.interact(productA1);
96          // Output:
97          // Interacting with ConcreteProductA1
98          // Using ConcreteProductA1
99
100         AbstractFactory factory2 = new ConcreteFactory2();
101         ProductA productA2 = factory2.createProductA();
102         ProductB productB2 = factory2.createProductB();
103         productB2.interact(productA2);
104         // Output:
105         // Interacting with ConcreteProductA2
106         // Using ConcreteProductA2
107     }
108 }
109
```

A real-world example is when there are two independent factories that can produce laptops and phones. Here, `productA = laptop` and `productB = phone`, but products at the first factory differ from product at the second factory.

Builder

Allows to produce different types and representations of an object using the same construction process. We extract the object creation code out of its class and move it to separate objects called *builders*.

- the builder has the same fields as the class
- the builder has setter-methods for every field and a `build()` method responsible for creating the class instance
- no-argument constructor is used for creating the builder

This is the basic idea:

```
1  public class Car {
2      private final String brand;
3      private final String color;
4      private final String model;
5
6      // constructor should be package-private of protected
7      Car(String brand, String color, String model) {
8          this.brand = brand;
```

```
        this.color = color;
        this.model = model;
    }
}



public class CarBuilder {
    private String brand;
    private String color;
    private String model;

    public CarBuilder brand(String brand) {
        this.brand = brand;
        return this;
    }

    public CarBuilder color(String color) {
        this.color = color;
        return this;
    }

    public CarBuilder model(String model) {
        this.model = model;
        return this;
    }

    public Car build() {
        return new Car(brand, color, model);
    }
}



class Main {
    public static void main(String[] args) {
        CarBuilder builder = new CarBuilder()
                .brand("Mitsubishi").color("red").model("b612");
        Car car = builder.build();
    }
}
```

Sometimes the same creation code is used to create several objects (for example, creating many `Bugatti` cars and many `Lamborgini` cars). In this case, we can use a **Director** -- a class which defines specific configurations depending on the case.

```java
public class Director {
    public void buildBugatti(CarBuilder builder) {
        builder.brand("Bugatti")
                .color("Blue")
                .model("bugatti model");
    }

    public void buildLamborghini(CarBuilder builder) {
        builder.brand("Lamborghini")
                .color("Yellow")
                .model("lambo model");
    }
}
```

Using a director is optional. Still, its advantage is that it completely hides the details of the product construction from the client code:

```java
public static void main(String[] args) {
    Director director = new Director();
    CarBuilder builder = new CarBuilder();

    director.buildBugatti(builder);

    Car car = builder.build();
}
```

Prototype

Every class that supports cloning is called a *prototype*.

The idea of the Prototype pattern is to delegate the object cloning process to the actual objects that are being cloned.

- the class should have a copy constructor and a `clone()` method overriden from the parent class
- the `clone()` method invokes the copy constructor
- the copy constructor both copies values of the class's fields and invokes the parent's copy constructor

```java
public abstract class Vehicle {
    private final String color;
    private final String brand;

    protected Vehicle(String color, String brand) {
        this.color = color;
        this.brand = brand;
    }
```

```
    // copy constructor – should be protected
    protected Vehicle(Vehicle other) {
        this.color = other.color;
        this.brand = other.brand;
    }

    public abstract Vehicle clone();
}
```

```
public class Car extends Vehicle {
    private final String model;

    public Car(String color, String brand, String model) {
        super(color, brand);
        this.model = model;
    }

    protected Car(Car other) {
        super(other);
        this.model = other.model;
    }

    @Override
    public Car clone() {
        return new Car(this);
    }
}
```

```
public class Bus extends Vehicle {
    private final int amountOfPlaces;

    Bus(String color, String model, int amountOfPlaces) {
        super(color, model);
        this.amountOfPlaces = amountOfPlaces;
    }

    Bus(Bus other) {
        super(other);
        this.amountOfPlaces = other.amountOfPlaces;;
    }

    @Override
    public Vehicle clone() {
        return new Bus(this);
    }
}
```

Shallow vs Deep copy

Imagine, we have fields of a reference type. For example, we use a class `Engine` as a field for a `Car` class:

```
public class Car extends Vehicle {
    private final String model;
    private final Egnine engine;

    // constructor

    protected Car(Car other) {
        super(other);
        this.model = other.model;
        this.engine = other.engine; // <------ shallow copy
        //          = other.engine.clone(); <------ deep copy
    }

    @Override
    public Car clone() {
        return new Car(this);
    }
}
```

- Shallow copy = we simply assign a value from the `other`'s field

```
public class Car extends Vehicle {
    private final Egnine engine;
    ...

    protected Car(Car other) {
        super(car);
        // assign values to other `Car` fields
        this.engine = other.engine;
    }
}
```

- Deep copy = we use `clone()` to get a copy of the `other`'s field

```
public class Car extends Vehicle {
    private final Egnine engine;
    ...

    protected Car(Car other) {
        super(car);
        // assign values to other `Car` fields
        this.engine = other.engine.clone();
    }
}
```

Structural patterns

Deal with inheritance and composition to provide extra functionality.

Behavioral patterns

Are about communication and assignment of responsibilities between objects.

NEED TO BE EDITED

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

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.

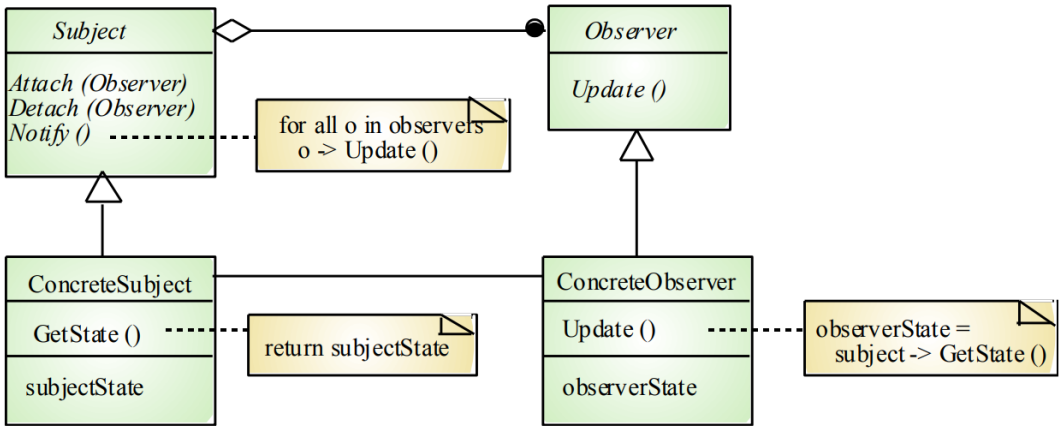
Object A above is called *subject*.

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

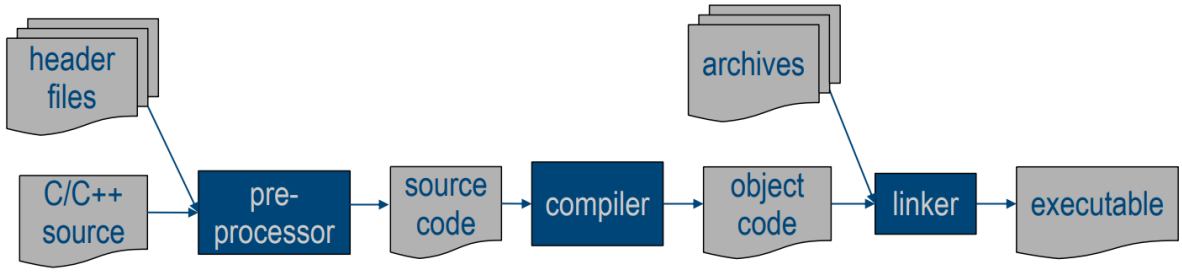
It is about composing a set of objects into a tree and then work with trees as if they were individual objects.

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

DOES NOT NEED TO BE EDITED ANYMORE

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:

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

23-03-21

Defensive Programming

Defensive programming = the creation of code for software designed to avoid problematic issues before they arise and to make the product more stable.

The basic idea behind this approach is to create a program that is able to run properly even when unforeseen processes or unexpected entries are made by users.

- Assertions, Exceptions, Return codes
- Loop invariants
- Class invariants:
    - constructors should create a new object in a valid state
    - methods should leave object in a valid state
- Method invariants:
    - users must meet pre-conditions
    - method guarantees post-conditions

23-03-28

Configuration management

Software Configuration Management (SCM) = the discipline of controlling the evolution of software systems

Revision — software object that was created by modifying an existing one

Variant — two software objects sharing an important property, differing in others

Version = Revision | Variant

Release — a version that has been made available to user/client

Configuration — selection of components from the repository that together make up a release

Three major configuration models:

1. Composition model (conf = set of software objects)
2. Change Set Model (conf = package of changes)
3. Long Transaction Model (conf = all changes are isolated into transactions)

23-04-18

Software testing

Software Testing = process of exercising a program in order to find errors before delivering it to the end user.

Developer understands the system but tests it "gently". Independent tester must learn about the system and attempts to break it.

Testing shows errors, requirements conformance, performance and the quality of the code (reliability, robustness, security, etc). The code with tests should be well structured and should have a documentation.

Source code and test code should be separated.

Equivalence Class Testing

Idea: build equivalence classes of input situations, test one candidate per class. Also, check boundaries: for boundary x check $x - 1$ and $x + 1$.

A "good" test case is when it is likely to produce an error.

Levels of testing

Unit testing — tests for every non-trivial function or method

Integration testing — check the integration of the application with all the outside components

End-to-end testing — check the entire system from start to end. performed under real-world scenarios, but they're expensive to perform and can be hard to maintain when they're automated.

UI testing — simulate user actions.

Beta testing — requires volunteers who will use the product for a while and point out its shortcomings.

Testing methods

Static testing

Collects information about a software without executing it (Reviews, walkthroughs, and inspections; static analysis; formal verification; documentation testing)

Static analysis -- control flow analysis and data flow analysis

Examples of errors that can be found:

- Unreachable statements
- Variables used before initialization
- Variables declared but never used
- Possible array bound violations

Dynamic testing

Collects information about a software with executing it

1. **White-box testing** -- looks inside module/class.
    - check that all statements & conditions have been executed at least once
    - check that no requirements are missing
2. **Black-Box testing** -- no knowledge about code internals, relying only on interface specifications.

    Limitations:

    - Specs are not always available
    - Many companies still have only code, there is no other document

Smoke testing

Executed to verify that critical functional parts are performing as expected and the whole system is stable. It makes no sense to send a program that has not passed this test for deeper testing.

Can be implemented at any level because the developers can make changes at any level.

Regression testing

Confirms that implemented changes have not negatively impacted the existing functionality/feature set.

Simply, it is about executing tests that already exist in order to check that the old code still works correctly.

Can be implemented at any level because the developers can make changes at any level.

23-05-25

Documentation

1. Internal — comments inside the code
2. External — separate doc about the code
3. User doc — explanation of what the code does from the user's view

Event-driven programming

Event — something happens (i.e. mouse motion). Event has properties (i.e. cursor position). Events form Event Queue.

Widget — a layer between the event queue and the code.

Event loop

```
while (app is running) {
  take event from event queue;
  pass it to the right widget;
}
```

MVC

MVC (Model-View-Controller) — an architecture for interactive apps

- *Model* — all data and logic
- *View* — visual display of the model
- *Controller* — receives input events from user and process them

UI design

Pressman's Golden Rules

1. **Place User in Control**: make the usage more comfortable for the user
    - avoid creating unnecessary interactions
    - allow user to interrupt or undo interactions
    - allow customizing the interface
    - hide technical internals
2. **Reduce User's Memory Load**: the user хочет запоминать о функционале как можно меньше
    - Reduce demand on short-term memory
    - provide meaningful defaults, undo and redo; visual cues; intuitive shortcuts
    - use real-world metaphors (words and actions should match the environment?)
3. **Make Interface Consistent**
    - put current task into meaningful context
    - if interaction has created expectations, do not change