Advanced Software Engineering Techniques

Course 2 – October, 16, 2017

Adrian Iftene
adiftene@info.uaic.ro
Content

- **Software Engineering Books**
  - GOF, GRASP, Swebok

- **Swebok**
  - Software configuration management
  - Software engineering management
  - Software engineering process
  - Software engineering tools and methods

**SOLID**
Software Engineering Books

- GOF (Gang–Of–Four) – Design Patterns: Elements of Reusable Object–Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vissides

- GRASP – Applying UML and Patterns – An Introduction to Object–Oriented Analysis and Design and Iterative Development, Craig Larman
Swebok project

- **Software Engineering Body of Knowledge**

- Book’s authors Alain Abran, James W. Moore, 2004
Guide to the Software Engineering Body of Knowledge (SWEBOK)

Get the 2004 SWEBOK Guide
- HTML (free)
- PDF
- Book

SWEBOK Guide Set for V3 Refresh

Volunteers are gearing up to refresh the Guide to the Software Engineering Body of Knowledge—SWEBOK—intending to add new knowledge areas (KAs) and to revise others.

Developed concurrently, the SWEBOK Guide, the Software Engineering 2004 (SE2004) curriculum guide, and the Certified Software Development Professional (CSDP) certification each provided a characterization of the discipline of software engineering. Despite nearly independent development, the three instruments agreed to a remarkable extent. The primary purpose of the current revision of the SWEBOK Guide is to perfect the correspondence between the three items, notably by adding a KA on professional practices—a subject currently covered by the CSDP—and adding “foundation” KAs on related subjects that software engineers learn about during their undergraduate education—subjects currently covered by SE2004.

To achieve this alignment and to maintain the currency of the SWEBOK Guide, the IEEE Computer Society’s Professional Practices Committee agreed in 2008 to the following changes:
- a new KA on Professional Practice—similar to material currently in the CSDP
Swebok – KAs

- The book is a Guide to the Software Engineering Body of Knowledge
- In the book are defined 10 *knowledge areas* (KAs) in SE
  - Software requirements
  - Software design
  - Software construction
  - Software testing
  - Software maintenance
  - Software configuration management
  - Software engineering management
  - Software engineering process
  - Software engineering tools and methods
  - Software quality
Software configuration management

- SCM – the task of tracking and controlling changes in the software
  - revision control and
  - the establishment of baselines

Q: "Somebody did something, how can one reproduce it?"
A: comparing different results and of analyzing their differences
SCM Definition

- *SCM is a supporting* software life cycle process (IEEE12207.0–96) which benefits project management, development and maintenance activities, assurance activities, and the customers and users of the end product.

- The concepts of configuration management apply to all items to be controlled (both hardware and software).

- SCM is closely related to the software quality assurance (SQA) activity.
SCM Activities

- Management and planning of the SCM process, software configuration identification, software configuration control, software configuration status accounting, software configuration auditing, and software release management and delivery.

---

**Figure 1. SCM Activities**

- **Coordination of Change Activities ("Code Management")**
- **Authorization of Changes** (Should changes be made?)
- **Status for:**
  - Project Management
  - Product Assurance
  - Development Team
- **Supports Customer Maintenance Team**
- **Physical & Functional Completeness**
- **Mgmt. & Planning**
- **Control Management Development Team**
- **Status Accounting**
- **Release Processing**
- **Auditing**

---

**SCMP**

---

**Configuration Identification**
RC also known as version control, source control or software configuration management (SCM) is the management of changes to documents, programs, and other information stored as computer files.

Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision"
SCM – Revision control (2)

- Each revision is associated with a timestamp and the person making the change.

- For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on.

- Revisions can be compared, restored, and with some types of files, merged.
Version control systems

- **Stand-alone applications**: Microsoft Word, OpenOffice.org Writer, KWord, Pages, Microsoft Excel, OpenOffice.org Calc, KSpread, Numbers

- **Content management systems**: Drupal, Joomla, WordPress

- **In wiki software packages** such as MediaWiki, DokuWiki, TWiki (offers the ability to revert a page to a previous revision. The aim is to correct mistakes, and defend public wikis against vandalism and spam)
SCM – Vocabulary (1)

- **Branch** – from that time forward, two copies of those files may develop at different speeds or in different ways independently.

- **Change/patch** – represents a specific modification to a document under version control.

- **Change list** – the set of changes made in a single commit.

- **Checkout** – creates a local working copy from the repository (we can specify a specific revision or obtain the latest).
SCM – Vocabulary (2)

- **Commit (checkin)** – occurs when writing or merging a copy of the changes made to the working copy into the repository
- **Conflict** – when different parties make changes to the same document, and the system is unable to reconcile the changes
- **Merge** – two sets of changes are applied to a file or set of files

![Diagram showing the workflow of checking out, checking in, and merging code changes between two developers.](image-url)
SCM – Vocabulary (3)

- **Tag** – refers to an important snapshot in time, consistent across many files. Can be a user-friendly, meaningful name or revision number.
- **Trunk** – The unique line of development that is not a branch.
- **Update** – merges changes from repository into the local working copy.
- **Working copy** – is the local copy of files from a repository.
Eclipse (1)
Eclipse (3)
Software engineering management (Swebok)

- The application of management activities—planning, coordinating, measuring, monitoring, controlling, and reporting—to ensure that the development and maintenance of software is systematic, disciplined, and quantified (IEEE610.12–90)
The perception of clients is such that there is often a lack of appreciation for the complexity inherent in software engineering.

It is almost inevitable to generate the need for new or changed client requirements.

Software is built in an iterative process rather than a sequence of closed tasks.

Software engineering necessarily incorporates aspects of creativity and discipline—maintaining a balance between the two is often difficult.

The degree of novelty and complexity of software is often extremely high.

There is a rapid rate of change in the underlying technology.
Figure 1 Breakdown of topics for the Software Engineering Management KA
Software Engineering Process (Swebok)

- Can be examined on two levels

- The **first level** encompasses the *technical and managerial activities* within the software life cycle processes

- The **second** is the *meta-level*, which is concerned with the definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle processes themselves
Figure 1 Breakdown of topics for the Software Engineering Process
Software engineering tools and methods (Swebok)

- **Tools** that are intended to assist the software life cycle processes, and to allow repetitive, well-defined actions to be automated, reducing the cognitive load.

- They are intended to make software engineering more systematic.

- **Methods** impose structure on the software engineering activity with the goal of making the activity systematic and successful.

- Methods provide a notation and vocabulary, procedures for performing identifiable tasks, and guidelines for checking both the process and the product.
Swebok – Related Disciplines

- Computer engineering
- Computer science
- Management
- Mathematics
- Quality management
- Software ergonomics (Cognitive ergonomics)
- Systems engineering
Computer engineering (Swebok)

- Computing Curricula 2001 project (CC2001) states that “computer engineering embodies the science and technology of design, construction, implementation and maintenance of software and hardware components of modern computing systems and computer-controlled equipment.”

- KAs for computer engineering:
  - Algorithms and Complexity
  - Computer Architecture and Organization
  - Computer Systems Engineering
  - Circuits and Systems
  - Digital Logic
  - Discrete Structures
  - Digital Signal Processing
  - Distributed Systems
  - Electronics
  - Embedded Systems
  - Human–Computer Interaction
  - Information Management
  - Intelligent Systems
  - Computer Networks
  - Operating Systems
  - Programming Fundamentals
  - Probability and Statistics
  - Social and Professional Issues
  - Software Engineering
  - Test and Verification
  - VLSI/ASIC Design
ISO Technical Committee 159 on Ergonomics as follows: “Ergonomics or (human factors) is the scientific discipline concerned with the understanding of the interactions among human and other elements of a system, and the profession that applies theory, principles, data and methods to design in order to optimize human well-being and overall system performance.”

KAs:

- Cognition
- Machine Learning and Grammar Induction
- Formal Methods in Cognitive Science: Language
- Formal Methods in Cognitive Science: Reasoning
- Formal Methods in Cognitive Science
- Information Extraction from Speech and Text
- Lexical Processing
- Computational Language Acquisition
- Human–Machine Fit and Adaptation
- Human Characteristics
- Computer System and Interface Architecture
- Dialogue Architecture
- Development Process
SOLID and Other Principles

- SOLID Principles
  - SRP – Single Responsibility Principle
  - OCP – Open/Closed Principle
  - LSP – Liskov Substitution Principle
  - ISP – Interface Segregation Principle
  - DIP – Dependency Inversion Principle
- DRY – Don't Repeat Yourself
- YAGNI – You Aren't Gonna Need It
- KISS – Keep It Simple, Stupid
SOLID was introduced by Robert C. Martin in the article called the “Principles of Object Oriented Design” in the early 2000s.

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle
SOLID – Single Responsibility Principle

- Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.
**SOLID – SRP – Definitions**

- “The Single Responsibility Principle states that every object should have a single responsibility and that responsibility should be entirely encapsulated by the class.” – Wikipedia

- “There should never be more than one reason for a class to change.” – Robert Martin

- Low coupling & strong cohesion
SOLID – SRP – Problems & Solutions

- Classic violations
  - Objects that can print/draw themselves
  - Objects that can save/restore themselves

- Classic solution
  - Separate printer & Separate saver

- Solution
  - Multiple small interfaces (ISP)
  - Many small classes
  - Distinct responsibilities

- Result
  - Flexible design
  - Lower coupling & Higher cohesion
SOLID – SRP – Example

- Two responsibilities

```java
interface Modem {
    public void dial(String pno);
    public void hangup();

    public void send(char c);
    public char recv();
}
```

- Separated interfaces

```java
interface DataChannel {
    public void send(char c);
    public char recv();
}
interface Connection {
    public void dial(String phn);
    public char hangup();
}
```
SOLID – Open/Closed Principle

- *Open chest surgery is not needed when putting on a coat*
- Bertrand Meyer originated the OCP term in his 1988 book, *Object Oriented Software Construction*
SOLID – OCP – Definitions

“The Open / Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” – Wikipedia

“All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version.” – Ivar Jacobson

- **Open** to Extension – New behavior can be added in the future
- **Closed** to Modification – Changes to source or binary code are not required
SOLID – OCP – How?

- Change behavior without changing code?!
  - Rely on abstractions, not implementations
  - Do not limit the variety of implementations

- In .NET – Interfaces, Abstract Classes

- In procedural code – Use parameters

- Approaches to achieve OCP
  - Parameters – Pass delegates / callbacks
  - Inheritance / Template Method pattern – Child types override behavior of a base class
  - Composition / Strategy pattern – Client code depends on abstraction, "Plug in" model
SOLID – OCP – Problems & Solutions

- Classic violations
  - Each change requires re-testing (possible bugs)
  - Cascading changes through modules
  - Logic depends on conditional statements

- Classic solution
  - New classes (nothing depends on them yet)
  - New classes (no legacy coupling)

- When to apply OCP?
  - Experience will tell you

- OCP adds complexity to design (TANSTAAFL)

- No design can be closed against all changes
// Open-Close Principle - Bad example
class GraphicEditor {
    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }
    public void drawCircle(Circle r) {
        ....
    }
    public void drawRectangle(Rectangle r) {
        ....
    }
}

class Shape {
    int m_type;
}

class Rectangle extends Shape {
    Rectangle() {super.m_type=1;}
}

class Circle extends Shape {
    Circle() {super.m_type=2;}
}

// Open-Close Principle - Good example
class GraphicEditor {
    public void drawShape(Shape s) {
        s.draw();
    }
}

class Shape {
    abstract void draw();
}

class Rectangle extends Shape {
    public void draw() {
        // draw the rectangle
    }
}

class Circle extends Shape {
    public void draw() {
        // draw the circle
    }
}
SOLID – Liskov Substitution

- If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction
- Barbara Liskov described the principle in 1988
"The Liskov Substitution Principle states that Subtypes must be substitutable for their base types." – Agile Principles, Patterns, and Practices in C#

- **Substitutability** – child classes must not
  - Remove base class behavior
  - Violate base class invariants

- **Normal OOP inheritance**
  - IS–A relationship

- **Liskov Substitution inheritance**
  - IS–SUBSTITUTABLE–FOR
SOLID – LSP – Problems & Solutions

The problem
- Polymorphism breaks Client code expectations
- "Fixing" by adding if–then – nightmare (OCP)

Classic violations
- Type checking for different methods
- Not implemented overridden methods
- Virtual methods in constructor

Solutions
- “Tell, Don’t Ask” – Don’t ask for types and Tell the object what to do
- Refactoring to base class – Common functionality and Introduce third class
// Violation of Liskov's Substitution Principle

class Rectangle{
    int m_width;
    int m_height;

    public void setWidth(int width){
        m_width = width;
    }

    public void setHeight(int h){
        m_height = h;
    }

    public int getWidth(){
        return m_width;
    }

    public int getHeight(){
        return m_height;
    }

    public int getArea(){
        return m_width * m_height;
    }
}

class Square extends Rectangle {
    public void setWidth(int width){
        m_width = width;
    }
}

    public void setHeight(int height){
        m_width = height;
    }
}

class LspTest
{
    private static Rectangle getNewRectangle()
    {
        // it can be an object returned by some factory ...
        return new Square();
    }

    public static void main (String args[])
    {
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10);

        // user knows that r it's a rectangle. It assumes that he's able to set the width and height as for the base class
        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
    }
}
SOLID – Interface Segregation

- You want me to plug this in. Where?
SOLID – ISP – Definitions

“The Interface Segregation Principle states that Clients should not be forced to depend on methods they do not use.” – Agile Principles, Patterns, and Practices in C#

Prefer small, cohesive interfaces – Interface is the interface type + All public members of a class

Divide "fat" interfaces into smaller ones
  ◦ “fat” interfaces means classes with useless methods, increased coupling, reduced flexibility and maintainability
Classic violations
  ◦ Unimplemented methods (also in LSP)
  ◦ Use of only a small portion of the class

When to fix?
  ◦ Once there is pain! Do not fix, if it is not broken!
  ◦ If the "fat" interface is yours, separate it to smaller ones
  ◦ If the "fat" interface is not yours, use "Adapter" pattern

Solutions
  ◦ Small interfaces
  ◦ Cohesive interfaces
  ◦ Focused interfaces
  ◦ Let the client define interfaces
  ◦ Package interfaces with their implementation
interface Worker {
    void work();
    void eat();
}

ManWorker implements Worker {
    void work() {...};
    void eat() {30 min break};;
}

RobotWorker implements Worker {
    void work() {...};
    void eat() {//Not Applicable for a RobotWorker};
}

interface Workable {
    public void work();
}

interface Feedable{
    public void eat();
}
SOLID – Dependency Inversion

- Would you solder a lamp directly to the electrical wiring in a wall?

Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?

Port doesn't define device
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.” – Agile Principles, Patterns, and Practices in C#
SOLID – DIP – Dependency

- Framework
- Third Party Libraries
- Database
- File System
- Email
- Web Services
- System Resources (Clock)
- Configuration

- The new Keyword
- Static methods
- Thread.Sleep
- Random
SOLID – DIP – Problems & Solutions

- **How it should be**
  - Classes should declare what they need
  - Constructors should require dependencies
  - Dependencies should be abstractions and be shown

- **How to do it**
  - Dependency Injection
  - The Hollywood principle "Don't call us, we'll call you!"

- **Classic violations**
  - Using of the new keyword, static methods/properties

- **How to fix?**
  - Default constructor, main method/starting point
  - Inversion of Control container
//DIP - bad example

class EmployeeService {
    private EmployeeFinder emFinder //concrete class, not abstract. Can access a SQL DB for instance
    public Employee findEmployee(...) {
        emFinder.findEmployee(...)
    }
}

//DIP - fixed

class EmployeeService {
    private IEmployeeFinder emFinder //depends on an abstraction, not on an implementation
    public Employee findEmployee(...) {
        emFinder.findEmployee(...)
    }
}

//Now it is possible to change the finder to be a XmlEmployeeFinder, DBEmployeeFinder, FlatFileEmployeeFinder, MockEmployeeFinder....
Other Principles

- Don't Repeat Yourself (DRY)
- You Ain't Gonna Need It (YAGNI)
- Keep It Simple, Stupid (KISS)
OP – Don't Repeat Yourself

- Repetition is the root of all software evil
OP – DRY – Definitions

- "Every piece of knowledge must have a single, unambiguous representation in the system.“ – The Pragmatic Programmer

- "Repetition in logic calls for abstraction. Repetition in process calls for automation.“ – 97 Things Every Programmer Should Know

- Variations include:
  - Once and Only Once
  - Duplication Is Evil (DIE)
OP – DRY – Problems

- Magic Strings/Values
- Duplicate logic in multiple locations
- Repeated if–then logic
- Conditionals instead of polymorphism
- Repeated Execution Patterns
- Lots of duplicate, probably copy-pasted code
- Only manual tests
- Static methods everywhere
OP – You Ain't Gonna Need It

- Don’t waste resources on what you might need.

"Always implement things when you actually need them, never when you just foresee that you need them. “ – Ron Jeffries, XP co-founder
OP – YAGNI – Problems

- Time for adding, testing, improving
- Debugging, documented, supported
- Difficult for requirements
- Larger and more complicated software
- May lead to adding even more features
- May not be known to clients
OP – Keep It Simple, Stupid

- You don’t need to know the entire universe when living on the Earth
"Most systems work best if they are kept simple." – U.S. Navy

"Simplicity should be a key goal in design and unnecessary complexity should be avoided." – Wikipedia
Links

- SOLID principles with real world examples: [http://blog.gauffin.org/2012/05/solid-principles-with-real-world-examples/](http://blog.gauffin.org/2012/05/solid-principles-with-real-world-examples/)
Bibliografie


- Erich Gamma, Richard Helm, Ralph Johnson, John Vissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addisson Wesley, 1998