Software System Design

16-lectures course presented by Yegor Bugayenko
to 3rd-year BSc students of Innopolis University in 2021,
and video recorded

The entire set of slide decks is in yegor256/ssd16 GitHub repository.

Thinking Behind the Course

The context in which the new course will be taught: There are 150 students on the course, one instructor, three teaching assistants, 16 lectures, 16 labs, 90 minutes per each lecture, presented on-site in the University, streamed online via Zoom, and later published on YouTube.

The context in which the new syllabus will be used: The audience consists of 3rd-year BSc students, who are mostly people with some practical experience of writing code. The purpose of the Syllabus is to manage their expectations and prepare for the examination.

The reasons why this new course and/or this new syllabus is needed and how it will fit into IU curriculum: As far as I understand, modern computer science education is strong in theories but lacks the connection with practical engineering. In other words, students have little chance of being taught by those who professionally write code every day and solve real-life technical problems. Me teaching them software design may be one of these opportunities: to link their theoretical learning with the practice.

The view(s) of learning in your course how learning happens in the course:
The course is all about problem-based learning. Many aspects of software design are decomposed into individual problems and solutions are discussed. Also, it is suggested to students to make their own projects and solve problems there.

The view(s) of learning assessment in the course: The assessment is based on software products, which students create. There are five criteria which are being assessed: requirements, design, architecture, code, and the "spirit" of development. *Formative* evaluation is only performed by TAs, three times during the course at alpha, beta, and release milestones: students present their software products and TAs subjectively evaluate them using the criteria define below. *Summative* evaluation is done by myself at the end of the course and is based on the evaluations provided by TAs three times during the course: my decision is also subjective, but preliminary evaluations help students understand pros and cons of their products. There is no significant difference between high and low stake assessments in the course, since all five criteria are balanced: they both are *middle stake* assessments.

The view(s) of progression in the course: At the end of each lecture I'm giving them suggestion of what they may try to use in their projects. I explain them why these "calls-to-action" may help them understand software design better. There are few progression scenarios expected: 1) from knowing **how to code** they will progress to knowing how to explain design decisions in UML, 2) from UML they will progress to knowing **how to use design patterns**, 3) from design patterns they will progress to knowing **how to test** their modules and automate tests, 4) from tests they will progress to **more complex decisions** about data, performance, formats, and interoperability.

The view(s) of recycling in the course: The course is teaching incremental and interactive programming, and it is also designed in incremental manner: every few weeks students must present their products and collect feedback. It is expected that their knowledge is recycled (refreshed) on each delivery cycle, since they have to review all layers of their products. They have to go through everything they know about software design a few times during the course.

The view(s) of alignment in the course: Learning **objectives**: by the end of the course students are able to design a software component and explain made design decisions. Learning **activities**: during the course students are instructed how to design a software and how design decisions can be explained, using different formats and approaches, such as UML, design patterns, RUP/SAD, and so on. Learning **assessment**: at the end of the course software products created by students are evaluated to check whether design decisions are sound and well explained.

The view(s) of course design framework you used/will use to design your course: Integrated course design framework is used to design this course. At initial design phase course objectives were defined. At intermediate design phase the body of the course was created. At the final design phase the course was re-evaluated for consistency.

The view(s) of course evaluation: I created a Telegram chat group for all students of the course and asked them to provide feedback after some lectures. The results collected (in form of anonymous polls) I used to correct myself and improve future lectures. For example, I gave more examples of practical software design decisions in response to negative comments about some of my slides. Another example, I formalized the Syllabus of the course in response to comments about its vagueness. I also had an additional synchronization session with TAs in response to complaints about our disintegration.

Potential risks in the course and the ways to address them: I was doing this course for the first time, that's why there were many risks. First, I expected myself being wrong at some points given to them. Second, I anticipated their lack of interest to certain topics inside the course. Third, I wasn't sure that the topics I cover are aligned with our courses. In order to mitigate these risks I organized a few workshops with TAs, who are more knowledgeable in this aspect. They helped me understand the context where my course is placed. Moreover, they gave me materials from previous year course of the same title — it was helpful.

Course Aims

Prerequisites to the course (it is expected that a student knows this):

- How to code
- How to use Git

After the course a student *hopefully* will know:

- How to manage software requirements
- How to develop iteratively and incrementally
- How to think with objects, not procedures
- How to use design patterns and not use anti-patterns
- How to draw and share knowledge using UML
- How to choose and use data formats, e.g. XML or JSON
- How to choose a database management server
- How to deploy software continuously
- How to build distributed software systems
- How to test software
- How to measure the quality of software design

They will also be able to:

- Document requirements in SRS and use cases
- Make key design decisions
- Explain them in UML
- Organize repository in GitHub
- Automate the build and cover it with tests

Assessment

At the end of the course a student receives a *score* of up to 100 points. The points are given after a *subjective* review of an open source software product created by the student during the course (no oral presentation is needed). Even though the review is subjective, the following balance has to be maintained (the questions provided below stand merely as examples and do not constitute the entire scope):

- **Requirements** (15%): Glossary is in place? Stakeholders and their concerns are identified? Use cases explain functional requirements? Non-functional requirements are documented? NFRs are measurable?
- **Design** (25%): UML diagrams, such as Class, Component, Deployment, and Sequence, are present? Design decisions are explained? Design patterns are used? Traceability between requirements and design elements is visible?
- **Architecture** (30%): The design is modular? The composition of modules makes sense? Design elements are cohesive? Design elements are decoupled enough? The build is automated? The delivery pipeline is automated?
- **Code** (15%): The code is clean enough? In-code documentation is present? Static analyzers and style checkers are used? Unit tests are in place? Integration tests are present? Is test coverage being measured?
- **Spirit** (15%): The product is somewhat popular on GitHub (or a similar platform)? Issues and pull requests were used during development? Commit comments are detailed enough? GitHub features are actively used, like releases, actions, etc.?

A few versions of the product may be presented for review: Alpha, Beta, and Final. The scores given to a student after version reviews don't affect the overall score given at the end of the course. However, if Alpha version is not delivered, a student gets a penalty of 10 negative points, while a missed Beta gives 20 negative points. Thus, if a student ignores both versions and brings a great product at the end of the course, he or she gets $100 - 30 = 70$ points at most.

The score may be turned into a grade using the following formula:

- **A** Excellent: 90+
- **B** Good: 75+
- **C** Satisfactory: 55+
- **D** Poor: 0+

Learning Material

The following books are highly recommended to read (in no particular order):

Len Bass et al., *Software Architecture in Practice*

Paul Clements et al., *Documenting Software Architectures: Views and Beyond*

Karl Wiegers et al., *Software Requirements*

Alistair Cockburn, *Writing Effective Use Cases*

Steve McConnell, *Software Estimation: Demystifying the Black Art*

Robert Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*

Steve McConnell, *Code Complete*

Frederick Brooks Jr., *Mythical Man-Month, The: Essays on Software Engineering*

David Thomas et al., *The Pragmatic Programmer: Your Journey To Mastery*

Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*

Grady Booch et al., *Object-Oriented Analysis and Design with Applications*

Bjarne Stroustrup, *Programming: Principles and Practice Using C++*

Brett McLaughlin et al., *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*

David West, *Object Thinking*

Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*

Yegor Bugayenko, *Elegant Objects*

Michael Feathers, *Working Effectively with Legacy Code*

Martin Fowler, *Refactoring: Improving the Design of Existing Code*

Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*

Scott Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*

Elliotte Rusty Harold et al., *XML in a Nutshell, Third Edition*

Michael James Fitzgerald, *Learning XSLT: A Hands-On Introduction to XSLT and XPath*

Martin Fowler, *UML Distilled*

Anneke Kleppe et al., *MDA Explained: The Model Driven Architecture: Practice and Promise*

C.J. Date, *An Introduction to Database Systems, 8th Edition*

Pramod Sadalage et al., *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*

Jez Humble et al., *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*

Michael T. Nygard, *Release It!: Design and Deploy Production-Ready Software*

Leonard Richardson et al., *RESTful Web APIs: Services for a Changing World*