LECTURE 1: INTRODUCTION TO SOFTWARE DESIGN

ANDREY.BOCHARNIKOV@GMAIL.COM

TELEGRAM: @RICKO_X

ARCHITECTURE

There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies.

C.A.R. Hoare (1985)

SOFTWARE LIFECYCLE

- Software concept
- Requirements Gathering & Analysis (preliminary)
- Design of the Architecture
- Design of the subsystems and components
- Implementation, Testing
- Deployment to Production
- Support. Address feedback. Next iteration

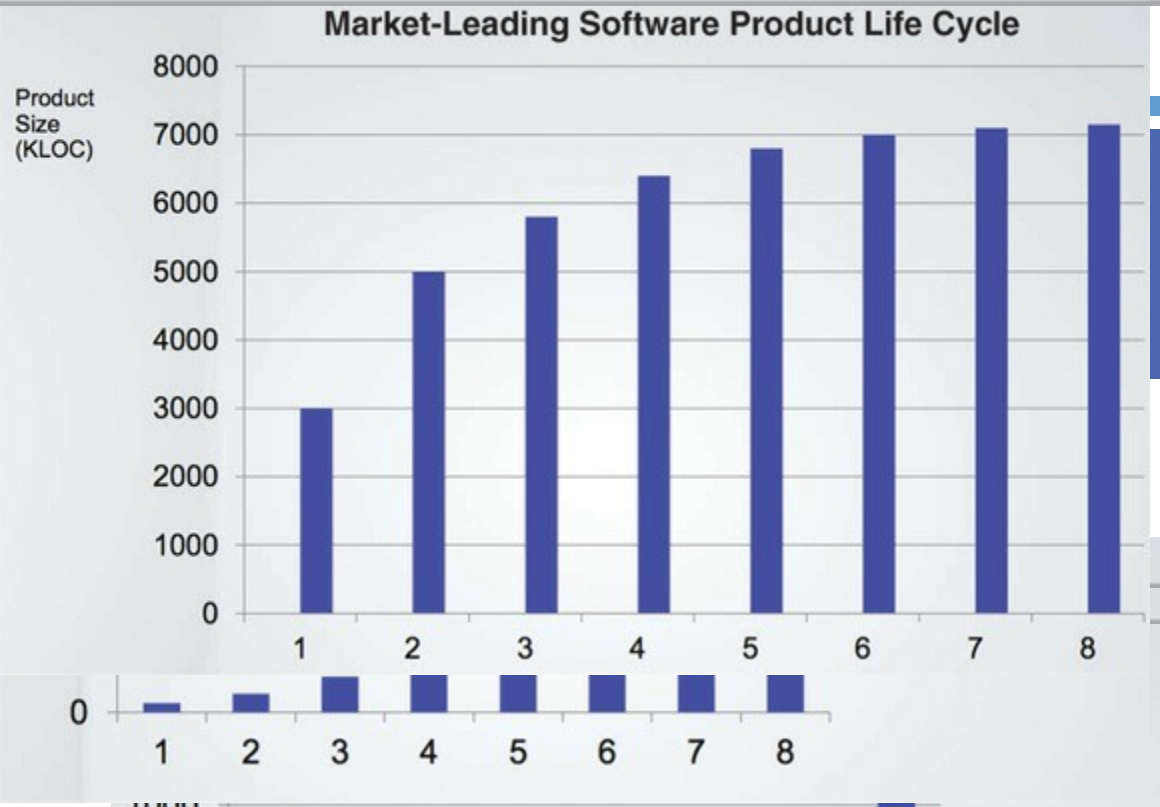
WHY

"Good software architecture makes the rest of the project easy."

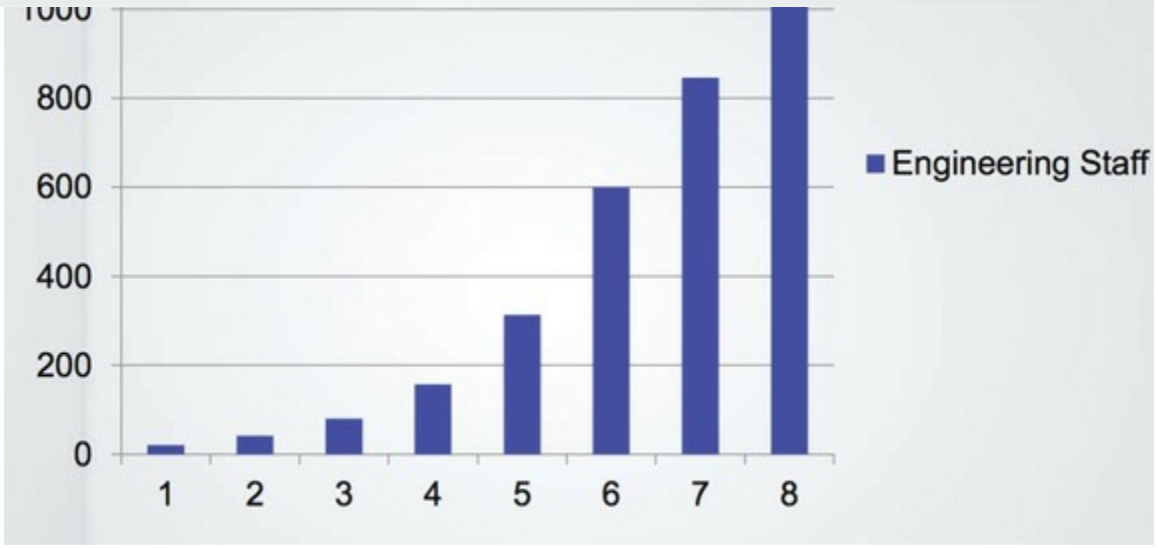
Steve McConnell, Survival Guide

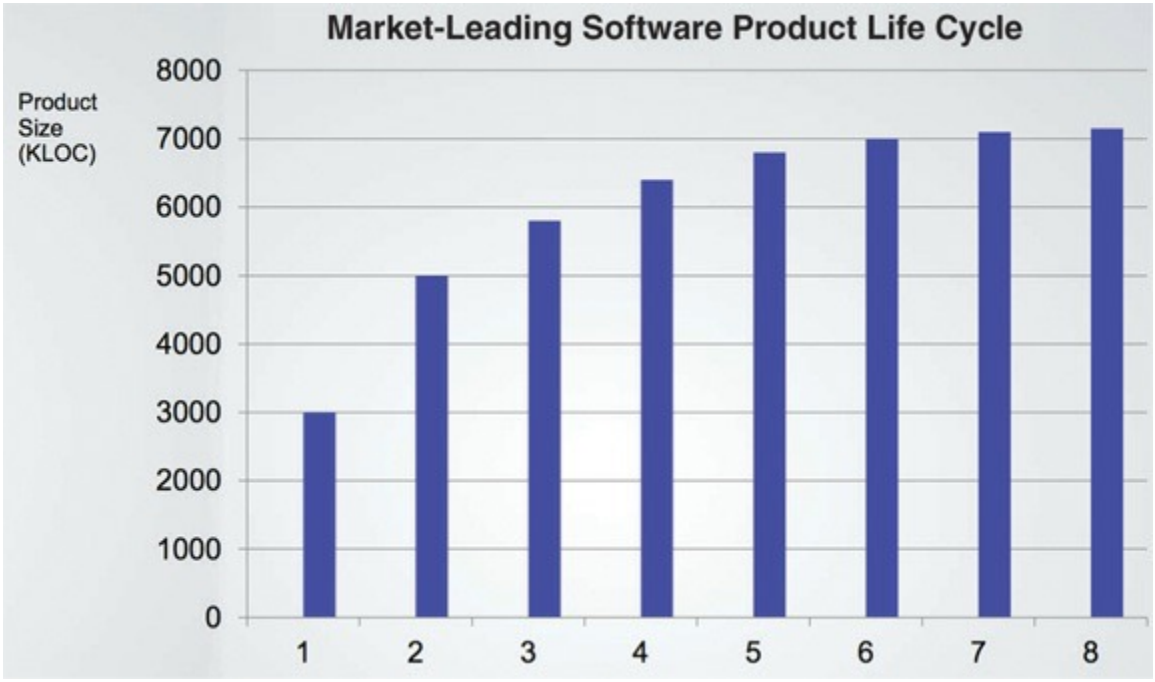
"The goal of software architecture is to minimize the human resources required to build and maintain the required system."

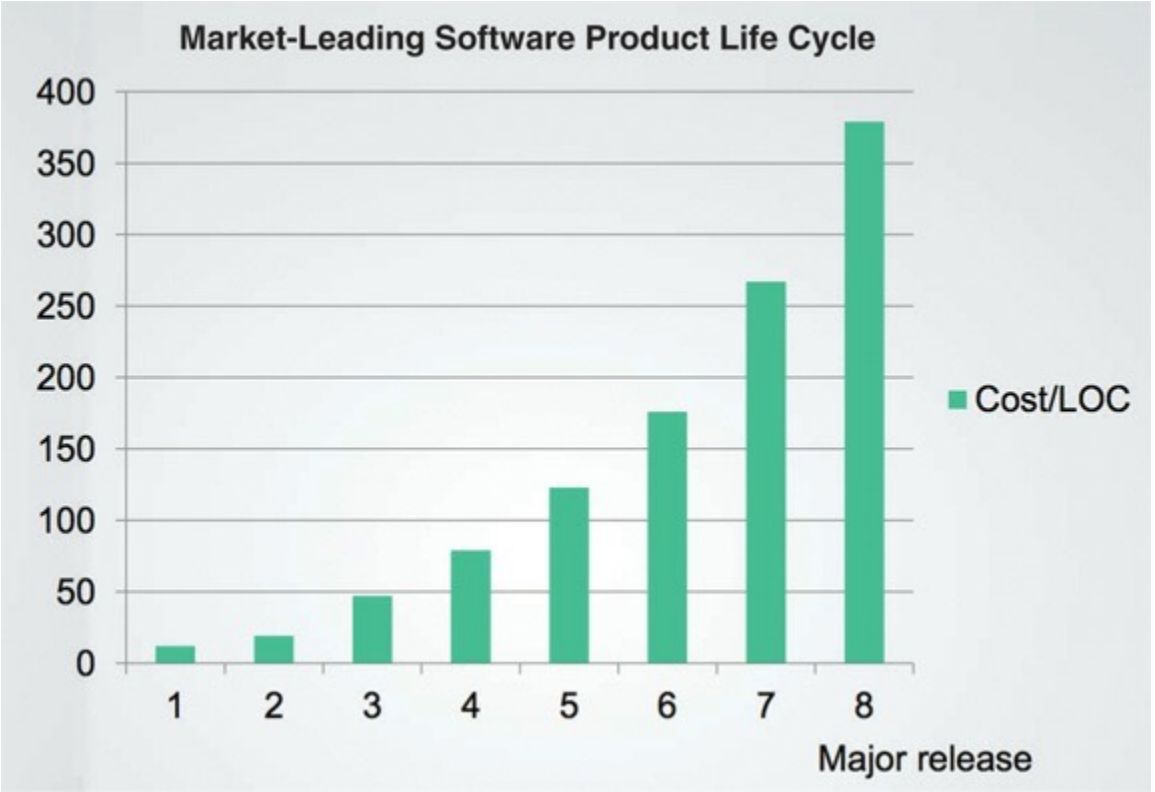
Robert Martin, Clean Architecture


Market-Leading Software Product Life Cycle
Product Size (KLOC)
8000
7000
6000
5000
4000
3000
2000
1000
0
1
2
3
4
5
6
7
8

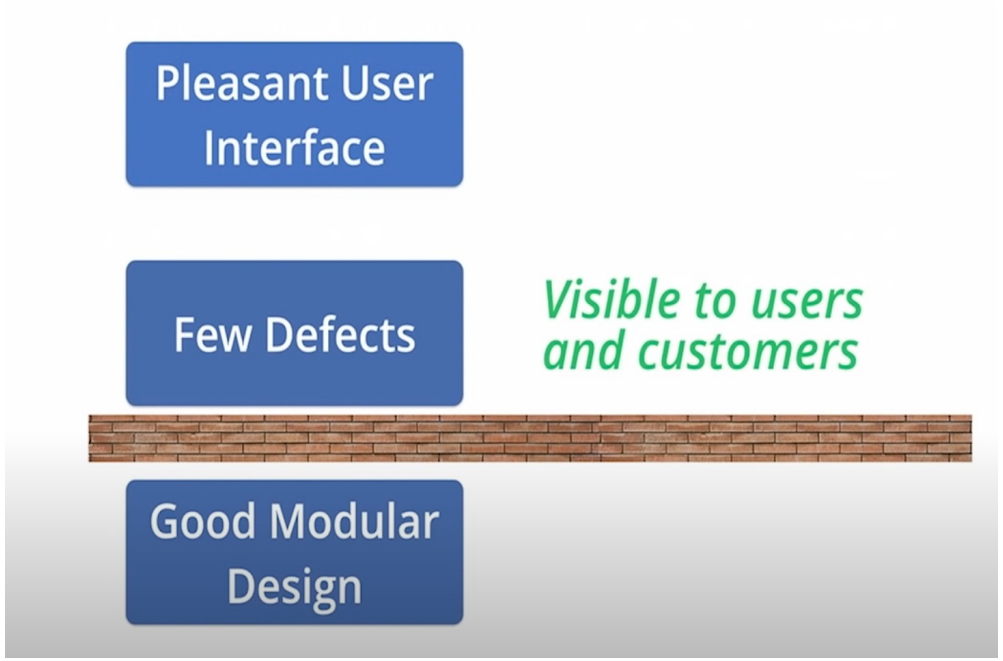
0
1
2
3
4
5
6
7
8


800
600
400
200
0
1
2
3
4
5
6
7
8
Engineering Staff

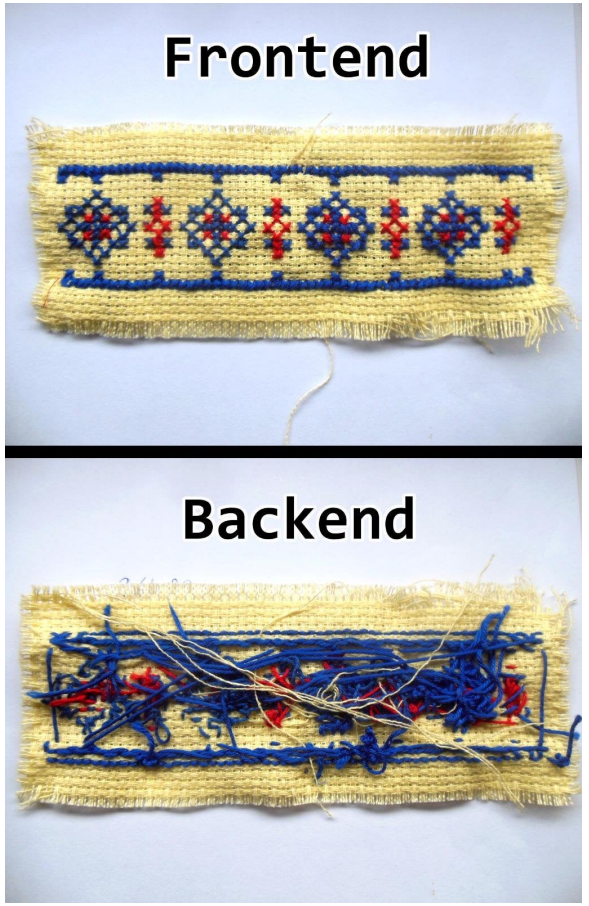

Market-Leading Software Product Life Cycle
Product Size (KLOC)
8000
7000
6000
5000
4000
3000
2000
1000
0
1
2
3
4
5
6
7
8


Market-Leading Software Product Life Cycle
400
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
8
Major release
Cost/LOC

GOAL


Pleasant User Interface
Few Defects
Visible to users and customers
Good Modular Design

vs


Frontend
Backend

GOOD ARCHITECTURE

- Satisfies functional and performance requirements
- Manages complexity
- Accommodates future change
- Is concerned with
  - reliability, safety, understandability, compatibility
- robustness …

REQUIREMENTS GATHERING

- Q&A
- UML
- Mind Maps
- Whatever you want

REQUIREMENTS CHECKLIST

- Specific Functional Requirements
  - Inputs/outputs to/from the system
  - Output formats for Web pages, reports, and so on
  - External hardware and software interfaces
  - External communication interfaces, hand- shaking, error-checking, and communication protocols
  - All the tasks the user wants to perform specified

REQUIREMENTS CHECKLIST

- Specific Nonfunctional (Quality) Requirements
  - Expected response time, from the user's point of view, specified for all necessary operations
  - Other timing considerations specified, such as processing time, data-transfer rate, and system throughput?
  - Level of security
  - Reliability, including the consequences of software failure, the vital information that needs to be protected from failure, and the strategy for error detection and recovery
  - Minimum machine memory and free disk space specified
  - The maintainability of the system
  - Is the definition of success included? Of failure?

REQUIREMENTS CHECKLIST

- Requirements Quality/Completeness
  - Does each requirement avoid conflicts with other requirements?
  - Do the requirements avoid specifying the design?
  - Are the requirements complete in the sense that if the product satisfies every requirement, it will be acceptable?
  - Are you comfortable with all the requirements? Have you eliminated requirements that are impossible to implement?

UML

Diagrams are another powerful heuristic tool. Everyone knows that it's better to see once than to hear a hundred times. Diagrams allow you to present a problem at a higher level of abstraction, and no description can replace them. Remember: sometimes a problem must be dealt with at a detailed level, and sometimes it is useful to deal with more general aspects

UML = universal modeling language

- A standardized way to describe (draw) architecture
  - Also implementation details such as subclassing, uses (dependences), and much more
- Widely used in industry

- Design Principles
- Design Patterns
- Design Approaches
- Best Practices
- Languages & Paradigms

DIVIDE AND CONQUER

- Benefits of decomposition:
  - Decrease size of tasks
  - Support independent testing and analysis
  - Separate work assignments
  - Ease understanding
- Use of abstraction leads to modularity
  - Implementation techniques: information hiding, interfaces
- To achieve modularity, you need:
  - Strong cohesion within a component
  - Loose coupling between components
  - And these properties should be true at each level

QUALITIES OF MODULAR SOFTWARE

- Decomposable
  - can be broken down into pieces
- Composable
  - pieces are useful and can be combined
- Understandable
  - one piece can be examined in isolation
- Has continuity
  - change in reqs affects few modules
- Protected / safe
  - an error affects few other modules

INTERFACE AND IMPLEMENTATION

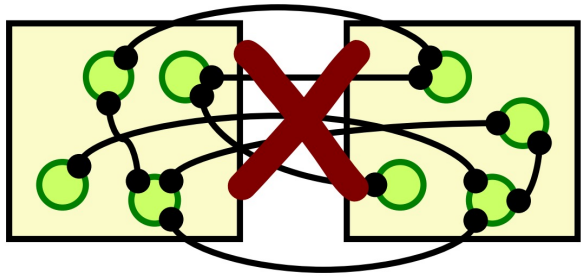
- Public interface: data and behavior of the object that can be seen and executed externally by "client" code
- Private implementation: internal data and methods in the object, used to help implement the public interface, but cannot be directly accessed
- Client: code that uses your class/subsystem
- Example: radio
  - public interface is the speaker, volume buttons, station dial
  - private implementation is the guts of the radio; the transistors, capacitors, voltage readings, frequencies, etc. that user should not see

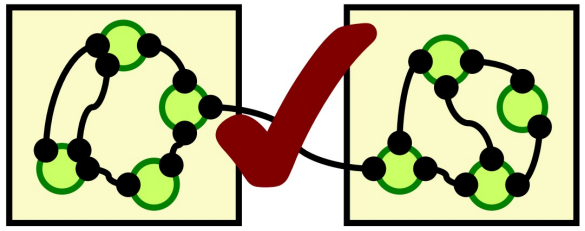
PROPERTIES OF ARCHITECTURE

- Coupling
- Cohesion
- Style conformity
- Matching
- Errosion

COUPLING (LOOSE VS. TIGHT)

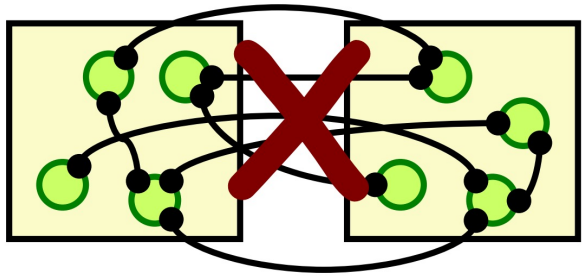
- Coupling: the kind and quantity of interconnections among modules
- Modules that are loosely coupled (or uncoupled) are better than those that are tightly coupled
- The more tightly coupled two modules are, the harder it is to work with them separately

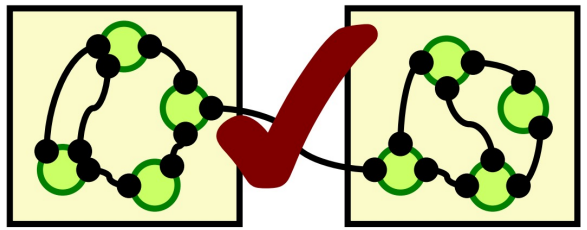




COHESION (STRONG VS. WEAK)

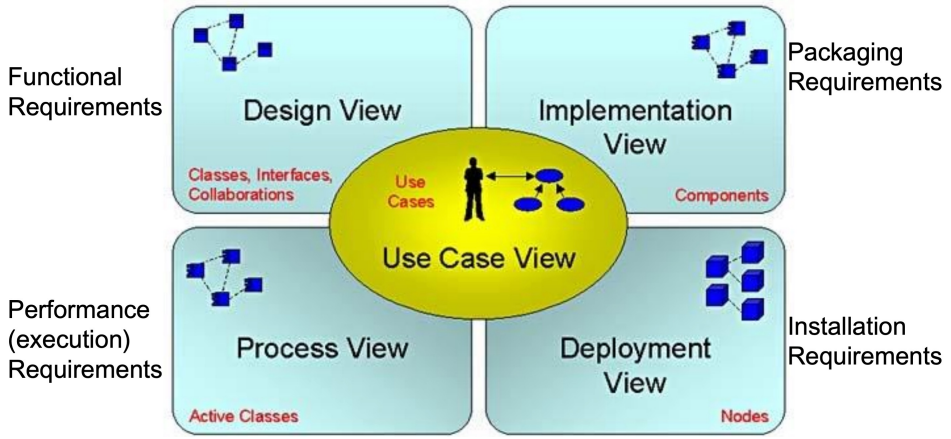
- Goal: the contents of each module are strongly inter-related
- High cohesion means the subcomponents really do belong together
- Tight relationships improve clarity and understanding
- Classes with good abstraction usually have strong cohesion





VIEWS

- A view illuminates a set of top-level design decisions
  - how the system is composed of interacting parts
  - where are the main pathways of interaction
  - key properties of the parts
  - information to allow high-level analysis
- Multiple views are needed to understand the different dimensions of systems



DESIGN PRINCIPLES

Design principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected. The use of the word "class" does not imply that these principles are applicable only to object-oriented software. A class is simply a coupled grouping of functions and data. Every software system has such groupings, whether they are called classes or not.

- SOLID
- GRASP
- YAGNI
- KISS
- DRY

PARADIGMS

- STRUCTURED PROGRAMMING, 1968
  - goto -> control structures such as if/then/else and do/while. Modules that used only those kinds of control structures could be recursively subdivided into provable units.
  - Structured programming allows modules to be recursively decomposed into provable units, which in turn means that modules can be functionally decomposed. That is, you can take a large-scale problem statement and decompose it into high-level functions. Each of those functions can then be decomposed into lower-level functions
- OBJECT-ORIENTED PROGRAMMING, 1966
  - Encapsulation (hide the data)
  - Inheritance
  - Polymorphism
- FUNCTIONAL PROGRAMMING, 1958

RECOMMENDED BOOKS

- “Clean Code” Robert Martin
- “Code Complete” Steve McConnell
- “Refactoring” Martin Fowler
- “Domain-Driven Design: Tackling Complexity in the Heart of Software” Eric Evans
- “Clean Architecture” Robert Martin
- “Domain Specific Languages“ Martin Fowler
- “Design Patterns: Elements of Reusable Object-Oriented Software” (aka “Gang of Four’s Book”) Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides
- “Functional Design and Architecture” Alexander Granin

CONCLUSION

- An architecture provides a high-level framework to build and evolve a software system.
- Strive for modularity: strong cohesion and loose coupling.