(An Introduction to) Software Architecture and Design

Maurício Aniche
m.f.aniche@tudelft.nl
@mauricioaniche
Delft University of Technology
For this lecture:

• “Go beyond the code”.
• Abstract thinking.

• *This lecture is compulsory if you want to become a software architect!*
Software Architecture is the **global organization** of a software system, including:

- the division of software into **subsystems/components**
- **policies** according to which these subsystems **interact**
- the definition of their **interfaces**.

T. C. Lethbridge & R. Laganière
The software architecture of a program or computing system is the structure or **structures of the system**, which comprise software **components**, the **externally visible properties** of those components, and the **relationships** among them.

Len Bass (Emeritus Professor, SEI)
The fundamental **concepts** or **properties** of a system in its **environment** embodied in its **elements**, **relationships**, and in the principles of its **design and evolution**.
Software architecture is driven by

• **Functional needs**
  • Your requirements.

• **Quality needs**
  • Non-functional requirements, e.g., performance, security.

• **Constraints**
  • Technical, legal, economic.

• **Architectural drivers** are the design forces that will influence the early design decisions the architects make.
Some quality attributes

• Performance: How fast does it respond?
• Availability: Is it available when and where I need?
• Usability: How easy is it for people to use?
• Interoperability: How easily does it connect to other systems?
• Integrity: How does it react against unauthorized access or data loss?
• Robustness: How well does it respond to unexpected conditions?
• Reliability: How long does it run before experiencing a failure?
• Recoverability: How quickly does it recover from a failure?
More quality attributes...

• Efficiency: How well does it use memory, disk space, ...?
• Flexibility: How easy is it to add new/extend functionality?
• Maintainability: How easy is it to fix bugs and make changes?
• Reusability: Can we reuse its components in other systems?
• Scalability: Can we add more users and servers?
• Testability: Can I easily verify that the system works correctly?
• ...
In practice...

- Different architectural styles address different sets of quality attributes.
- Not all quality attributes are addressed by architectural design.
  - E.g., usability might be better addressed by UX design, or performance by algorithms.

- Impossible to maximize all attributes at once!
  - Trade-offs are common in practice!
Case study: edX grading tool

- Students will watch videos and see exercise descriptions at edX.
- Students will have different coding exercises to solve. Each exercise has a different way to be automatically graded / feedback.
  - We should provide students with feedback about what they did wrong in the exercise.
  - We should send the results (grade) of a graded assignment back to edX.
- Students will use GitHub/GitLab to store their code.
- Final grades + feedback should also be stored at TU Delft.
- We expect 10-20k students.
  - Remember that students tend to submit assignments near the deadline.
- Teachers need analytics.
  - We should see most common mistakes, harder exercises, ...
student

does coding exercises

watches

GitHub

Notify about an event

Send feedback

Send grade + feedback

teacher

Sees reports and analytics

Send grade to edX

grader

Notify about an event

Send feedback

Put grade on edX
GitHub

POST event (JSON)

API Gateway

Queue the event

SQS

webhooks
Poll from the queue

Remove item from the queue

Clone student code

Test grader

Safe environment

Comment in the Github’s PR

Feedback to TUD
Poll from the queue

Monitors SQS and increase number of Docker instances
This is software architecture!

• We discussed how:
  • Systems will communicate
  • How they will scale
  • How they will be deployed
  • How our users will interact with the system

• We could have used a more formal notation...
  • UML can help us on that.
  • See activity, communication, component, and deployment diagrams.
Architectural patterns?

• There are many of them.
• Different patterns help you achieve different quality attributes
  • Performance
  • Scalability
  • ...

• In this course, we do not study them in details. 😞
Architecture is strategic, while Design is tactical.

Architecture comprises the frameworks, tools, programming paradigms, component-based software engineering standards, high-level principles..

While design is an activity concerned with local constraints, such as design patterns, programming idioms, and refactorings.
Software Design

• Tactical decisions.
• Long list of software design principles

• Grady Booch: Abstraction, Encapsulation, Modularization, and Hierarchy are the four fundamental software design principles.  
  Booch, Grady; et al. (2004). Object-Oriented Analysis and Design with Applications (3rd ed.).

• Robert Martin talks about design odors: rigidity, fragility, immobility, viscosity, needless complexity, needless repetition, opacity.  
Abstraction

• What’s the high-level concept behind it?
• What are the important and unimportant properties?
• What do components share in common?

• More formally: “An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer” (Grady Booch)

• Good abstractions reduce coupling!
Order
- totalAmount
- amountLimit
+ calculateTaxes()

Item
- product
- quantity
- price

Discount
- apply()

Christmas Discount

Carnival Discount
Item
- product
- quantity
- price

Order
- totalAmount
- amountLimit
+ calculateTaxes()

Discount
- apply()

Christmas Discount

Carnival Discount

Represents the **Order (concept)**
What all Discounts have in common?
Cohesion

• The degree to which elements belong together.

• In other words: “Does it have a single responsibility, or does it do more than one thing?”
  • You may replace “it” by your function, your file containing many functions, classes, an entire system...

• Non-cohesive elements tend to be:
  • Harder to be reused.
  • More complex, and thus, harder to maintain.

• Thus, we want elements to be as much cohesive as possible.
High cohesion

Low cohesion
What would make these abstractions less cohesive?
Coupling

• How close/connected two elements are?

• “A needs B”. Again, replace A and B by any element in your system:
  • My function a() invokes b(), so A and B are coupled.
  • If I import file A, I should import file B too. So, they are coupled.
  • If I want to use system A, I also need to deploy system B. So, they are coupled.

• Highly coupled components are:
  • Hard to reuse
  • Fragile

• We want our elements to be less coupled.
High coupling

Low coupling
Coupling and cohesion seem very related to each other!
Circular dependencies

• Circular dependencies are:
  • Often complex
  • Highly coupled
  • Hard to test

• Might be a problem for some compilers...

• Careful: we are not talking about recursive algorithms here.
Information Hiding

• Hiding away internal details about our components.
  • Again, component being a function, a module, a system...

• In practice:
  • What does X do? ← You should be able to answer that!
  • How does X do its job? ← You should not be able to answer that!
  • What is inside, must stay inside.

• Components not well encapsulated are hard to be changed!
  • You probably have to propagate a change in different places.
  • Easy to break / make mistakes.
More formally

- The purpose of **information hiding** is to obtain a modularization of the code of a system that isolates changes into single modules.
  - David L. Parnas. "On the criteria to be used in decomposing systems into modules", CACM, Dec., 1972.
  - Nice read: [https://blog.acolyer.org/2016/09/05/on-the-criteria-to-be-used-in-decomposing-systems-into-modules/](https://blog.acolyer.org/2016/09/05/on-the-criteria-to-be-used-in-decomposing-systems-into-modules/)

- **Encapsulation**: "the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation." (Grady Booch)
In practice: How do I check for encapsulation?

• Using my test code!

• I write tests to my function, and:
  • If I can guess “how” my function does its job without having to go to its real implementation, then, my function is not well-encapsulated.
  • In this case, I refactor.

• Not writing tests? Have always two files in your IDE
  • The one you are implementing.
  • Another one that calls the functions you are implementing.
Modularity

• Divide a system in a way that each part can be designed and revised independently.

• Modular systems enables us to:
  • Reduce complexity (two small modules are maybe less complex than one big module)
  • Easier to test
  • Easier to “throw it away” / be replaced

• Grady Booch: “Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.”
My entire software
(order, deliveries, reports)
becomes

Order

Delivery

Reports
My entire software (order, deliveries, reports)

Modules are small, cohesive, and easier to be replaced

becomes

Order

Delivery

Reports
Designing modular systems is challenging

- Modules should hide information.
- Modules should expose as little as possible.
- Modules should cooperate.

- Again, the coupling vs cohesion battle.
Hierarchy

• Define how different abstractions are related to each other.
  • A Course has many Lectures.
  • A Teaching Assistant is a User with High Privileges.

• More formally, “Hierarchy is a ranking or ordering of abstractions.” (Grady Booch)
Hierarchy principles

• Open/Close principle (OCP)
  • Modules should be open for extensions.
  • “It should be easy for a developer to extend/add new functionality”.
  • “Developers don’t have to change too much code for that. Even better, don’t have to just add new code, and not change existing code”.

• Dependency Principle (DIP)
  • Bob Martin: 1) “High-level modules should not depend on low-level modules. Both should depend on abstractions.”, 2) “Abstractions should not depend on details. Details should depend on abstractions.”
Item
- product
- quantity
- price

Order
- totalAmount
- amountLimit
+ calculateTaxes()

Discount
- apply()

- Christmas Discount
- Carnival Discount

Has many

High-level abstractions (Discount)
and low-level details (Christmas, Carnival)
Software design odors

- Rigidity – The design is hard to change
- Fragility – The design is easy to break
- Immobility – The design is hard to reuse
- Viscosity – It is hard to do the right thing
- Needless complexity – Overdesign
- Needless Repetition – Copy/paste
- Opacity – Difficult to read and understand
Empirical research

• Coupling, cohesion, and complexity rarely follow a normal distribution in a software system. Often they follow power laws, or pareto distributions (Herraiz et al, 2011), (Clauset et al, 2009).
  • This means most of our code is simple (coupling, cohesion, ...).
  • But the tail is long: complex code exists (coupling, cohesion, ...).
• Higher coupling, low cohesion have been related to change- and defect-proneness (Briand et al. 1998, Shatnawi and Li 2009, Eski and Buzluca, 2011, ...)
  • This means coupling and cohesion matter!

Empirical research

• The context and architecture of the software system matters (Gil and Lalouche, 2016).
  • Programming language
  • age

• Some components are naturally more coupled, cohesive, and complex than others, and this should be taken into account when analyzing your software (Aniche, 2016).
  • What “low coupling” means might vary...


Summary

• Software architecture
• Software design
Software architecture:

• provides a **communication** among stakeholders
• captures **early design decisions**
• acts as a transferable **abstraction** of a system
• defines **constraints** on implementation
• dictates **organizational structure**
• inhibits or enables a system’s **quality attributes**
• is analyzable and a vehicle for **predicting** system qualities
• makes it easier to reason about and **manage change**
• enables more accurate **cost** and **schedule** estimates
Software Design:

• Coupling vs Cohesion

• Grady Booch: Abstraction, Encapsulation, Modularization, and Hierarchy are the four fundamental software design principles.
  Booch, Grady; et al. (2004). Object-Oriented Analysis and Design with Applications (3rd ed.).

• Robert Martin talks about design odors: rigidity, fragility, immobility, viscosity, needless complexity, needless repetition, opacity.
“One of the differences between building architecture and software architecture is that a lot of decisions about a building are hard to change. It is hard to go back and change your basement, though it is possible. There is no theoretical reason that anything is hard to change about software. If you pick any one aspect of software then you can make it easy to change, but we don’t know how to make everything easy to change. Making something easy to change makes the overall system a little more complex, and making everything easy to change makes the entire system very complex. Complexity is what makes software hard to change. That, and duplication.”

-- Ralph Johnson

Getting your hands dirty

• In your project, there’s probably one very good improvement you can make, from the design perspective:

Separate your UI from your business logic!
Curious?

- SOLID principles
- Domain-Driven Design
Credits

• Initial discussion about Software Architecture was highly based on Imed Hammouda’s slides on Software Architecture: http://www.cse.chalmers.se/edu/year/2015/course/EDA222/Documents/Slides/Software_Architecture_IH.pdf

• Influences:
  • The Software Reengineering course, given by Andy Zaidman.
  • Lecture on Principles of Software Design, by Tip and Weintraub at the Northeastern University: https://course.ccs.neu.edu/cs5500sp17/designPrinciples.pdf
License

• You can use and share any of my material (lecture slides, website).
• You always have to give credits to the original author.
• You agree not to sell it or make profit in any way with this.

• Material that I refer has its own license. Please check it out.
Images in this presentation

• Building, by buggy: https://openclipart.org/detail/21547/build
• Question guy, by Scout: https://openclipart.org/detail/191766/question-guy
• Scale, by scott_Kirkwood: https://openclipart.org/detail/24101/scales
• Green tick, by jean_victor_balini: https://openclipart.org/detail/17056/tick
• Empty box thinking, by bsantos: https://openclipart.org/detail/5173/empty-box-thinking
• Owlies, by SHUI: https://openclipart.org/detail/291674/little-owl-friends