(Practical) SW Architectures

November 14, 2013
## SW Architecture

- The structure of an SW system.
- Separation of a software system into its components.
- Fundamental organization of the components.
- Relationships between these components, how they interact.
- Relationships to the environment (other systems like OS).
- Principles guiding the *design* of software.
- Principles guiding the *evolution* of architecture and software.
- Patterns, guidelines, processes, rules, methodologies, domains, languages.

- Not just a collection of boxes and lines.
- Not just a static view of an intended state.
- Not only the very-high-level structure.
- Not only an upfront design.
Why we need software architecture

- Why not just start coding and let things turn out somehow.
  - Lean models.
  - Agile development.
  - Quick iterations.
  - No upfront design.
- Software is increasingly complex.
- Software is *evolving*, reused in new contexts.
- The promise of off-the-shelf components with easy reuse is just partial.
- Sometimes we need to adapt, wrap or even modify these off-the-shelf components.
- Too many components in software products, small and large.
- Large number of connections/dependencies between these components.
- How to make sure that SW development reaches the intended goals of a project.
Why we need software architecture

- Software is complex:
  - >10KLOC (KLOC = 1000 lines of code): internal projects, tools, hobby projects, minimum viable products.
  - >100KLOC: small products, mobile apps, applications.
  - >1MLOC: operating systems, native frameworks, typical desktop software applications, server side applications, typical everyday product.

- How to understand software that has >1MLOC?
- How to reach >1MLOC and still understand software?
- How to reach >1MLOC without excessive complexity?
- How a >1MLOC software evolves into the next version?
- How a >1MLOC software or its components are reused (pivoted)?
What is the structure of an SW system

- What are the components of the system:
  - Can be a simple boxes and lines view.
- There can be multiple views of a system structure.
- The component structure is one of these views (logical or development).
- The components define their interfaces.
- The structure defines an external view of these components.
- Also defines how the interfaces are connected.
- Defines the dependencies between components.
- Various modeling languages and tools can be used to specify the structure.
- The structure is not static: may evolve during development.
- The structure also evolves during the product lifecycle.
- Principles to guide the evolution of the software structure.
Selection of SW components

- What are the components (small or large) of an SW system.
- First: identify the domains and the problems to solve.
- Define components based on these problems and domains.
- Define what is expected from these components by others.
- Define the interface of the component.
- The interface can be formulated based on the following:
  - The problem solved by the component.
  - The domain into which the problem belongs.
  - Always study solutions in the identified domain.
  - What is expected from the component by the system.
  - Off-the-shelf components have their interfaces defined.
Component design

- A component may define multiple interfaces.
- Off-the-shelf components define their interfaces based on the domain and problem they solve.
- This can be generic and sometimes complex.
- This may not always match the exact need of the project.
- Solution: *Adapter* pattern:
  - Given a *base* component whose interface needs to be adapted.
  - Define new interfaces based on what is expected from the component.
  - Implement a new *adapter* (or *wrapper*) component having the new interface.
  - The adapter component uses the base component in its implementation.
  - Other components see an external view of the new adapter component.
  - Other components can use the new interface.
Component design

- Off-the-shelf components are not always available or suitable for a given project.
- It is typical of a (large) software project to contain multiple new components.
- These components are implemented in the software project.

Architecture Guideline
A software project implementation should consist of a set of generic reusable component implementations.

Component Design Guideline
Component interfaces should be minimal and simple, nevertheless designed based on the domain and problem they solve. This applies to internal components also. If needed, use adapter interfaces for a specific access to the component.
Component reuse

Reusable Components
A software system should consist of a set of reusable components.

- Not everything can be formulated as a reusable component.
- Still a good architecture should converge to a collection of such components.
- Minimize the amount of glue code.
- Why? *Glue* code tends to become disorganized and often requires refactoring.
- Make it small, to minimize these refactoring efforts.
- A component (implementation) can be refactored if its interface does not change.
- A large amount of *glue* code that is not structured into components is harder to refactor (internal dependencies are not through interfaces).
A system is defined in a context of an environment.
The environment can be an operating system.
Or a specific product using the software.
Company/business model is an environment.
The goals of the company/business is an environmental factor.
The environment places limits on the architecture and influences requirements of the architecture.
The architecture also influences the environment:
- Is the business viable long term.
- Can you pivot in case of wrong product-market fit.
- How the software will evolve with various versions.
- What are the growth needs and possibilities of the company.
- Can you make derivative products.
Business driven requirements

- The type of the business and product influences architectural design.
- Is it a product or a platform.
- Is there a public API that can be used to extend the product.
- Role of software in the business:
  - Is software the main product of the company.
  - Is it a component in an other type of product.
  - Or an internal system with small or no public visibility.
- Lifecycle expectations: typically software is iterated through multiple versions.
- Scalability:
  - Number of users accessing simultaneously.
  - Amount of data processed by a single user.
- Pivoting: especially for startups, but also in case of larger companies.
Other choices of the architecture

- The choice of programming language is an architectural choice:
  - What type of software product is developed.
  - Desktop, mobile, native or web.
  - Static or dynamically typed languages.
  - Strong or weakly typed languages.
- Choice of frameworks used in the software:
  - User interface frameworks.
  - Database engines.
- Security: what are the architecture level decisions that improve security.
- Security requirements also determine the development processes, and may pose restrictions on component interfaces.
- Coding style: also important to be able to hand over code between developers.
What programming language(s) to use in the project?

The choice of language has multiple impacts:
- Typically strong personal preferences (opinions).
- The number of available skilled persons.
- The speed of development.
- The performance of the final product.
- Scalability and robustness.
- What works in the specific domain.

Large projects may use multiple languages at different levels of the system.

Ability to optimize low-level time critical routines.

Ability to quickly reconfigure high level features and user interface.
Type systems: an area each software architects should be familiar with.

Programming language type system:
- Has impact on both performance and development speed.
- Correlates with suitability for various domains problems.
- Is the language native, interpreted or runs on a VM or with JIT compiler.

Static vs. Dynamic type systems.
Weak vs. Strong type systems.
Weak + Static typing: C, parts of C++ and Objective-C.
Efficient but harder to work with.
Weak + Dynamic typing: specific uses of Objective-C and JavaScript.
Strong + Static typing: C++, Java.
Strong + Dynamic typing: Objective-C, Lisp derivatives, Python.
Principles guiding the design of software

- What is the architectural style used.
- The architectural style guides how to design the components.
- Individual components may have their architectural style internally:
  - As long as this does not conflict with the system architecture.
  - As long as component reuse is possible.
- The architecture defines architectural patterns for components.
- Define patterns for component interfaces.
- SW design patterns used in component implementations.
- Architecture is hierarchical:
  - Component internals can be isolated from the encapsulating architecture.

Guideline

A software architecture should be hierarchical. A simple and easy to understand high level view. Details on individual components can be explored hierarchically.
Principles guiding the evolution of software

- A software product/code always evolves.
- The development of the very first version is a process of evolution.
- How does the software evolve?
  - An important practical aspect of the software architecture.
- The most likely cause of failure of a software project or product:
  - Cannot evolve to the required direction.
- Evolution of software:
  - From idea to prototype.
  - From prototype to product.
  - From buggy to stable.
  - From basic to improved versions.
  - Evolving into derivative products.
  - Pivoting the product in case of wrong market fit.
The evolution of software

Guideline

The flexibility to **evolve** and **improve** is one of the most important attributes of a good software architecture.

- Software development occurs in a changing environment.
- Initial requirements may change during the development process.
- New requirements can show up, or existing ones dropped.
- How much impact this has on various software components.
- Requirements can be outside of the control of involved parties.
- Market needs can change requiring new type of user interactions.
- Ability to use existing components to quickly respond to emerging market needs.
- This is where lean and agile meets upfront architecture.
The evolution of software

- Do not just define *boxes and lines*.
- Define patterns and software design guidelines.
- Define interfaces on how to add extensions.
- Design interfaces so that they are stable against changing implementation.
- Example: a UI framework API should not depend on the style of the UI.
- In large software consider plugin architectures: ability to add/remove features easily.
- Changing environment: customer requirements, new markets, competitors.
- If components are reusable, they can be reused under changing environment.
- Short agile iterations are compatible with upfront architecture.
- If upfront architecture is about frameworks, design styles, patterns, and rules. Not so much about boxes and lines.
The evolution of software

- One aspect that differentiates successful companies from failed ones.
- How well they evolved their existing solutions into new products.
- Software is large and complex.
- Difficult to restart from the beginning for each new product type.
- With well designed reusable components.
- New product types can be easier to create.
- Companies (whose software architecture can evolve) can afford to create new product types and new markets.
- And these are not minimum viable products, but complex fully developed systems.
- This gives them extra lead over competitors.
- Especially if competitors software cannot evolve into the new market.
Other attributes of SW architecture

- A good architecture can be documented in terms understandable by a trained engineer.
- Concepts and metaphors introduced by the architecture must be defined.
- A good architecture is easy to communicate (through training and documentation).
- A good architecture does not hinder progress of the SW development.
- Easily understood and assimilated by practicing developers.
- Interfaces defined for components are stable.
- Changes to interfaces can be implemented with manageable impact.
- A good architecture employs orthogonal design.
Orthogonality

**Guideline**

Software architecture should consider orthogonality of design and features.

- What is orthogonality in the context of software design?
- Depends on the context and feature.
- The concept is similar to orthogonality in programming language design.
- Languages: a relatively small set of instructions combined in a number of ways to build the control and data structures in a language.
- In general: try to avoid repeating yourself.
- Different components should not implement the same feature repeatedly.
- Instead use a common component providing that feature.
- Combine multiple smaller independent components.
Orthogonality

- Orthogonality reduces complexity of software components.
- Orthogonality increases reuse of code.
- Typically, orthogonality may increase the number of components, while reducing their size.
- For example styling of UI elements.
- Wrong: each UI element implements its own styling mechanism.
- Good: implement a common styling engine that is used by the base UI component.
- Specific UI components then can have little or no styling related code.
Good architectures

Good Architecture

One measure of a good architecture is the how long it survives as a solution to a particular problem and in how many context can be used without changes.

- There is no clear, unified, objective way to define what is a good architecture.
- There are good and bad practices.
- There are patterns and anti-patterns.
- If a solution is good, usually survives the test of time.
- There are always good examples to get inspiration from.
- Solutions that have been around for a long time.
- Solutions that work in multiple contexts and environments independent of the product type.
Examples of good architectures

- Good architectural styles exits now in most fields.
- Pipes are a good model of combining multiple modules in processing data.
- Pipes can be extended into flow-based programming.
- Relational databases are a stable solution of a large group of database problems.
- The filing system is a good example of stable architecture:
  - In early computing it was not clear how to access/organize persistent data.
  - The hierarchical filing system solves this problem.
  - A small, clearly defined and easy to use interface.
  - Interface allows arbitrary implementation: various file systems focusing on speed, failure tolerance, etc.
- For user interfaces the Cocoa (NextStep) UI framework provides a good inspiration.
- POSIX is a good interface for OS designs.
Pipeline

- The *Pipes and Filters* architecture style/pattern.
- Models data flow between modules.
- Initially used to chain command line programs.
- The output of a program is the input of the next program.
- A common pattern used to combine program components.
- Practical and simple model that is used in modern architectures.
- Can be generalized into flow-based programming.

![Figure: Chain of filters processing data flow.](image-url)
Flow-based programming

- Flow-based programming.
- A generalization of pipes and filters.
- Used for example in image processing domain: node based composition.
- Construct a directed graph of software components.
- Preferably acyclic, but cyclic models can also be implemented.
- Nodes are software components having multiple inputs and one output.
- Inputs and outputs are also represented as nodes.
- Edges represent flow of data between components.
- Components collect data from inputs or other nodes through incoming edges.
- Components output data through one (or more) outgoing edges.
- Not just for image processing and compositing.
- Can model arbitrary flow between a set of components.
Figure: Flow-based programming is a generalization of pipes and filters.
Figure: Examples of Quartz Composer.
Node based composition

Figure: Node based composition is an application of flow-based programming.
Designing good architectures

- How to create/design good architectures?
- Keep in mind the attributes of a good architecture.
- It should simplify the system and make it easier to understand.
- Should guide through the steps of development.
- Large volume of existing solutions.
- Study solutions for various domains and problems.
- Learn existing architectural patterns.
- Software design patterns are also part of architecture (the Gang of Four book).
- Do not reinvent the wheel, improve on existing models.
- Simple designs usually are more resilient.
A set of basic principles of object-oriented design.
Should be part of the guiding principles of a software architecture.
One method to keep the design of a software clean.
Does not guarantee clean software by *blind* use.
SOLID:
- **S**: *Single responsibility principle*.
- **O**: *Open/closed principle*.
- **L**: *Liskov substitution principle*.
- **I**: *Interface segregation principle*.
- **D**: *Dependency inversion principle*.
Typically used in test-driven development, agile and adaptive models.
Should also guide good software architecture design.
Another set of tools: GRASP (not discussed in this lecture).
S: Single Responsibility Principle

- A class should have a single responsibility.
- This extends to components also, with a suitable definition of responsibility.
- All the interfaces/services provided by the class/component are driven by this responsibility.
- The class (or component) entirely encapsulates the responsibility.
- R.C. Martin defines the responsibility as a reason to change.
- In practice a responsibility is aligned with a specific problem in a domain.
- A component solves a specific problem.
- Multiple classes provided by the component deal with various responsibilities related to the problem.
In the pipeline pattern typically a node has a single task to perform.

Consider a UI framework with its components. The basic visual component is a view in Cocoa (or a widget in Qt).

The responsibility is to manage a rectangular area of the screen. This responsibility implies multiple tasks. What is drawn in this area and how to interact with it is not specified.

These base classes (NSView or QWidget) have large interfaces. Could also be separated into smaller responsibilities. But that would make internals of the problem public.

The definition of responsibility must be balanced:
- Large enough to hide internals of the problem.
- Small enough to keep the interface simple.
O: Open/Closed Principle

- Book: Bertrand Meyer, Object Oriented Software Construction.
- Software entities, classes or components should be:
  - Open for extension.
  - Closed for modification.
- The component should allow its behavior to be modified without altering the source code.
- Code should be designed extensible (Open).
- Problems are solved by extending and not modifying existing components (Closed).
- Practical note: *Code should also be written so that is modifiable.*
- For example bugs, or changes in the base problem formulation are still solved by fixing the internals of the component.

Open/Close Principle

The main goal of the open/close principle is to drive problem solution towards extending existing components instead of modifying them.
A component following the open/closed principle does not change.

This reduces the chance to introduce new bugs and the need for additional testing.

One of the likely source of bugs in software:
- Modifying existing components without understanding their internals.

One mechanism of extension is inheritance.

The implementation of the base class is extended by inheriting it.

There are other methods for modifying code behavior.

Injection is one such method (see the dependency injection pattern).

Aspect oriented programming also augments existing code.

Objective-C implements categories to extend existing classes without inheritance.

Other techniques are dynamic method bindings and delegates.
Liskov Substitution Principle

- Introduced by Barbara Liskov in 1987.
- If $S$ is a subtype of $T$, then instances of $T$ can be replaced with $S$ without altering the desirable properties of the program.

Liskov Substitution Principle

Let $q(x)$ be a property provable about object $x$ of type $T$. Then $q(y)$ should be provable for objects $y$ of type $S$, where $S$ is a subtype of $T$.

- This is a *strong* principle and in practical implementation the *desirable properties* is subject of interpretation.
- It is also called *strong behavioral sub typing*.
- For example, in practice $q$ is be a correctness property.
- Tasks performed may be specialized by sub typing, correctness not.
- Note: $q$ is a property of a base type $T$. The descendant contains a specialization of this property $q$. 
L: Liskov Substitution Principle

There are a number practical implications of the Liskov substitution principle.

First: public methods of $T$ are well defined and correct in $S$ also, since $S$ is a subtype of $T$.

It is considered a design error where a method $m$ of $T$ is not defined, incorrect or should not be used on instances of $S$.

Second: components using instances of $T$ should work as well with instances of $S$.

Let $C$ be a component that was implemented with the knowledge of the interface of $T$.

$C$ operates on instances of $T$.

Then $C$ should be able to operate as well on instances of $S$.

The result (performed task) might be different, but the program should be well defined and correct.
I: Interface Segregation Principle

- No client should depend on interfaces it does not use.
- This principle drives towards small and focused interfaces and components.
- Reduces the dependency of component towards other components.
- In theory only interfaces that are used by a component should be imported.
- In practice this is not always possible.
- A component may define and publish a set of interfaces.
- If dependency is a relation between components (not just classes), then client components may have visibility towards interfaces that are not used.
- However, in practice this does not create significant problems.
I: Interface Segregation Principle

- Both the *interface segregation* and the *single responsibility* principles drive the architecture towards focused problem definition.
- A problem should be small enough to be solvable.
- Small problems are easier to formulate and solve.
- Nevertheless, a problem formulation should not be a restrictive (special) case.
- This usually results in additional similar components (different cases of the problem).
- And the decision of which of those components to use is a leaking of the internals of the problem.
- A problem solution when used by clients should not expose additional problems to the clients.
- A client should not depend on the internal problems of a component.
- Internal dependencies of a component are not visible to a client.
D: Dependency inversion principle

- 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.
- In dependency inversion the low-level components depend on interfaces defined by higher level components.
- The interface towards a low level component is defined by the higher level component.
- The low level component then implements this interface.
- Why? High level components define *what is required.*
- Instead of building on (low level) interfaces available at the time.
- Define interfaces towards low level layers.
- These interfaces are restricted and specific to the high level component.
- Thus this interfaces can be smaller/simpler than a general purpose abstraction layer.
Dependency inversion principle

Figure: (a) component dependency. (b) dependency inversion.
Dependency inversion principle

- Adapters can be implemented that couple the interfaces defined by high level components with actual interfaces available towards low level components.
- Replacing low level components is easier since the higher level components don’t depend on these interfaces.
- Model also defined by the adapter pattern.
- Patterns like plugin or dependency injection can also be used.
- For example a high-level document editor loads/stores documents.
- The format(s) in which the document are stored is implemented different components.
- Some of these formats are 3rd party components.
- Different formats have different APIs/interfaces.
- The high level document defines an interface what is expected of a format plugin.
- Multiple format plugins then can be implemented and selected for loading or storage.
Software is important

- Software becomes increasingly more important in everyday products.
- Not just computers or consumer electronics.
- Cars, medical devices, airplanes, financial systems, trading platforms.
- The role of software increases in products that were previously strictly hardware.
- For example there are a growing number of companies developing in-car software systems.
- Also the performance and correct function of modern cars depend on increasingly complex software.
- Yesterdays mobile companies must become tomorrows software companies.
- If they can’t then yesterday software companies take over the market (Google, Apple).
- Software failures can cause significant damage.
- Failure to successfully develop software can also ruin large companies.
Anti-patterns

- It is also important to know what does not work.
- Where are the pitfalls of software development.
- Anti-patterns collect such models of software development.
- Typically show up in failed software projects.
- Better to learn these by reading them instead of experiencing them.
- A successful software project is a *continuous detection and removal* of emerging anti-patterns.
- An evolving software can and will go *bad* over time.
Some examples of anti-patterns

- **Stovepipe system**: ad-hoc architecture, subsystems are integrated using multiple strategies. Software structure that inhibits change. Not to confuse with organic/dynamic software structure.
- **Feature creep**: an ongoing addition of features to compensate for product quality. Typically features have impact on a large amount of components.
- **Vendor lock-in**: software is highly dependent on some proprietary technology. A typical cause of failure in software businesses. Isolation layers help to solve this.
- **Warm Bodies**: some programmers are more productive than others. Teams are built to meet staff size objective, not qualification objectives.
- **Design by Committee**: creates overly complex architectures that lack coherence and flexibility. More persons do not substitute for talent.
- **Swiss Army Knife**: lack of focus in component/problem definition.
- **Reinvent the Wheel**: no software reuse, lack of knowledge on existing solutions.
Can an architecture or software code go *bad*?

Also called *code rot, software erosion*.

Accumulating *Technical Debt*.

Software is not static, it undergoes continuous development.

Even for software that has a single release cycle.

The development process is a process of change.

What is the direction of this change: improvement or erosion.

It is the second law of thermodynamics applied to the software development process.

Unless extra effort is invested to continuously improve, software will erode over time.

Since software evolves (through development), it is typical that erosion occurs.

This can be prevented by good architecture and processes.
Software erosion can be avoided through continuous improvement.

Software development is also a process of learning.

Lessons learned can be incorporated into the architecture and processes.

This includes both patterns and anti-patterns.

Components/interfaces can be refactored when new knowledge is available.

Refactoring is simplified by a good architecture.

Refactoring is even easier if interfaces can be stable.

Interfaces may also need refactoring.

Using dependency inversion can reduce the amount of change caused by interface refactoring.

Steps of improvement (refactoring) should be part of any software project.
Signs of erosion

- Architectural rule violations: bypassing the processes or patterns specified by the architecture.
- Cyclic dependencies: A depends on B depends on C depends on A.
- Dead code: code that is not used anymore but still in the system.
- Code clones: the result of the copy-paste anti-pattern. Nearly identical code cluttered across the system. Change in one case will result in change of all instances.
- Metric outliers: deep class hierarchies, huge packages, complex code, large methods, functions or interfaces.
- Leaking abstractions: internals of the implementation are visible to the user of a component.
- Workarounds: as the project progresses, more and more problems are solved through workarounds and quick fixes.
- Proliferation of special cases: more and more special cases show up in components.
- Undocumented dependencies: developers tend to use existing components for solving problems those components not meant to solve.
Steps to avoid erosion

- A well designed architecture with patterns, guidelines, rules and processes that are suitable for the given project.
- Use dependency inversion when cyclic dependencies show up.
- Remove code that is not needed anymore.
- Never copy-paste: formalize problems, define an interface and implement it. Then allow dependencies to this new component when needed.
- Study and refactor metric outliers.
- Avoid leaking abstractions by designing interfaces that simplify a specific domain/problem.
- Avoid workarounds: find the root cause of a problem. Sometimes more immediate work, but less damage later.
- If there are many special cases, maybe the problem definition is wrong.