Software System Architecture and Design

The Evolution Perspective
The Evolution Perspective

- **Desired Quality**: The ability of the system to be *flexible* in the face of the *inevitable change* that all systems experience after deployment,
  - balanced against the costs of providing such flexibility.
- **Applicability**: Important for all systems to some extent;
  - more important for longer-lived and more widely used systems.
The only constant is **change**, and most software architects agree strongly with this.

Common factors such as:
- misunderstood requirements,
- rapid business change, and
- the effect that actually delivering a system has on end-user requirements,

and it is easy to see why **change is such a major factor**.

When a system is delivered in **iterations**, its users can start using some parts of it much earlier and thus provide **early feedback** to the developers.

- It means that there is **constant pressure** during the delivery cycle to change the system’s behavior, with a consequent need in some cases to change its architecture.

Experienced software developers agree that software is easy to change only if **change was explicitly considered** during its development.

- Software developed without any concern for the changes can be much harder to change than anyone expects.
### Applicability to Views

<table>
<thead>
<tr>
<th>View</th>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>If the evolution required is significant, the functional structure will need to reflect this.</td>
</tr>
<tr>
<td>Information</td>
<td>If information evolution is needed, a flexible information model will be required.</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Evolutionary needs may dictate particular element packaging or some constraints on the concurrency structure (e.g., that it must be very simple).</td>
</tr>
<tr>
<td>Development</td>
<td>Evolution requirements may have a significant impact on the development environment that needs to be defined (e.g., enforcing portability guidelines).</td>
</tr>
<tr>
<td>Deployment</td>
<td>Rarely significant impact</td>
</tr>
<tr>
<td>Operational</td>
<td>Rarely significant impact</td>
</tr>
</tbody>
</table>
Concerns: Product Management

- In a product development environment, the evolution of a product is usually planned and overseen by a product manager,
  - who is responsible for understanding the needs of the product’s customers and the threats and opportunities offered by the market in which the product is sold.
- The product manager’s job is:
  - to use this insight to define a roadmap for the future development of the product and
  - to work with the product development organization in order to deliver and evolve the product in line with it.
- Product management provides direction for all of the change happening in the system.
Concerns: Dimensions of Change

- **Functional evolution**: This includes any *change to the functions* that the system provides, from simple defect corrections to the addition or replacement of entire subsystems.

- **Platform evolution**: Many successful systems need to evolve in terms of the *software and hardware platforms* on which they are deployed. This can include:
  - migrating platforms (e.g., from Windows-based servers to Linux-based servers)
  - extending the platforms the system can use (e.g., porting products to new platforms or extending existing PC-based client platforms to include Web-based access).

- **Integration evolution**: Most information systems need to be integrated with a number of other systems to be useful.
  - As these other systems are created, evolve, and are removed, this may put evolution pressures on your system – it may need to *change the way it integrates* with other systems.

- **Growth**: Most successful systems undergo a growth in usage during their lifetime.
  - This may be due to many factors, such as an increase in the number or complexity of transactions, an increase in the number of users, or the need to manage and store larger amounts of data.
Applying the Evolution Perspective

1. Characterize Evolution Needs

2. Assess Current Ease of Evolution

3. Consider Evolution Tradeoffs

[finished]

[not finished]

4. Apply Tactics to Rework the Architecture
1. Characterize the Evolution Needs

- **Type of change required**: Characterize each type of evolution into one of the dimensions described earlier (functional, platform, integration, or growth).
- **Magnitude of change required**: Establish how much effort each type of evolution will need. Is it just defect correction, or will large-scale, high-risk system changes be required?
  - A useful way to present this is the effort required as a proportion of the initial system development effort.
- **Likelihood of change**: Assess how likely it is that each of the types of change you have identified will actually be required.
  - This allows you to focus on those that are most likely to occur.
- **Timescale of required changes**:
  - Are the changes required on an immediate, firm timetable (in effect, a phased delivery)?
  - Or are they vague needs for changes sometime in the future depending on external factors (such as system growth)?
2. Assess the Current Ease of Evolution

- For each of the evolution requirements you have identified, work through a scenario of how you would change your system to meet the requirement when it becomes necessary.
- For each of these scenarios, note:
  - how much of the system must change and
  - how difficult and risky the resulting set of changes is.
- This assessment allows you to decide whether or not your architecture requires any changes to meet the evolution requirement.
3. Consider the Evolution Tradeoffs

- The key **tradeoff** to consider is:
  - whether to expend the effort of creating a flexible system **during initial development** or
  - whether to **defer** this effort until system changes are actually required.

- This tradeoff depends very much on:
  - the **type** of system,
  - the **likelihood** of changes actually being required, and
  - the **level of confidence** you have about easily making major changes when needed, rather than during initial development.

4. Rework the Architecture

- Use the best evolution strategy you identified to make the set of changes necessary for your architecture to support the evolution requirements.
Architectural Tactics: Contain Change

- Typically, dealing with change in a small, well-defined part of your system isn’t a problem.
- A change starts to be a problem when its effects ripple through a number of different parts of the system simultaneously.
- The architectural challenge is to design a system structure so that the required changes are as contained as possible.
- Design principles (actually mandated by the Functional Viewpoint!):
  - *Separation of concerns*
  - *Encapsulation*
  - *Single point of definition* – DRY (Do not Repeat Yourself)
  - *Functional cohesion*
  - *Low coupling*
  - *Abstract common services*
How to achieve low coupling

- Techniques to achieve low coupling are:
  - Introduce **stable** interfaces between communicating components – ripple effect of changes stops at the interface boundary
  - Code that **creates** class/component instances is moved away from classes/components
    - Design patterns – Abstract Factory, Builder
    - Dependency Injection (DI) techniques – EJB, CDI, SpringFramework
    - Stable interfaces are still mandatory!
  - Remove the **dependency** between the two components
    - Event-driven techniques (CDI)
    - Stable interfaces are still mandatory (event realizes the interface)!

- Where we need to have low coupling?
  - Not between all the classes/components!
  - Modules should be decoupled
    - Modules are defined in the Development Viewpoint.
Create Extensible Interfaces

- Changes to interfaces usually have the widest impact and are therefore the most costly.

- Some techniques:
  - Replace APIs that have large numbers of individual parameters with ones that pass in objects or other structured data types instead.
    - For example, a `CreateEmployee` method might take as input the employee first and last name, date of birth, and Social Security number.
    - This could be replaced by a method that takes an `Employee` object instead.
    - When members are added to the `Employee` class, they can be given appropriate defaults so that code does not need to be changed everywhere that the method is invoked.
  - You can use a similar approach with *information interfaces*.
    - For example:
      - by using a self-describing message technology such as XML to define message formats, and
      - allowing new message elements to be optional, you can allow messages to be extended with little or no impact on system elements that do not need to use the extended form of the interface.
Apply Design Techniques That Facilitate Change

- There are a number of architectural styles that can help make a system more amenable to change. Some of them:
  - **Abstraction and layering** patterns make it easier to change one part of the system with minimal impact on others.
  - **Generalization** patterns make it easier to handle new use cases or data types, by just specializing the existing general-purpose functionality in a way that is appropriate to the new use case.
  - **Inversion of Control** (also known as “dependency injection”) and **Callback** patterns help protect higher-level elements in your architecture from the implementation specifics of lower-level elements.
Apply Metamodel-Based Architectural Styles

- Metamodel-based systems provide a very high degree of flexibility in some problem domains.
- Metamodel approaches:
  - break down the system’s processing and data into their fundamental building blocks and
  - use runtime configurations to assemble these into fully functional components.
- Changes to requirements can often be made by changing the metamodel, rather than having to change the underlying software components.
- Examples:
  - Expression evaluation: configuration file contains grammar definition, system invokes grammar compiler
  - Report generators
  - MDA – Model Driven Architecture
- Metamodel-based systems are much more complex to develop and test than systems based on more static architectural styles.
- They are also inherently less efficient in terms of runtime performance, which can limit their applicability in environments where performance is a major concern.
Build Variation Points into the Software

- A large number of specific software design patterns have been published that attempt to introduce some form of variation point.

<table>
<thead>
<tr>
<th>What varies</th>
<th>Design Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithms</td>
<td>Strategy, Visitor</td>
</tr>
<tr>
<td>Actions</td>
<td>Command</td>
</tr>
<tr>
<td>Implementations</td>
<td>Bridge</td>
</tr>
<tr>
<td>Response to change</td>
<td>Observer</td>
</tr>
<tr>
<td>Interactions between objects</td>
<td>Mediator</td>
</tr>
<tr>
<td>Object being created</td>
<td>Factory Method, Abstract Factory, Prototype</td>
</tr>
<tr>
<td>Structure being created</td>
<td>Builder</td>
</tr>
<tr>
<td>Traversal Algorithm</td>
<td>Iterator</td>
</tr>
<tr>
<td>Object interfaces</td>
<td>Adapter</td>
</tr>
<tr>
<td>Object behavior</td>
<td>Decorator, State</td>
</tr>
</tbody>
</table>
Use Standard Extension Points

- Many mainstream information systems technologies provide standard extension points.
- Example: Java EE platform allows to easily add support for:
  - new types of databases, via the JDBC interface,
  - external systems, via the JCA interface, etc.
Defer binding time

- **Deferring binding time** allows non-developers to make changes at the cost of requiring additional infrastructure to support the late binding.

- Many techniques allow deferring binding time:
  - **Runtime registration** supports plug-and-play operation at the cost of additional overhead to manage the registration. Publish/subscribe registration, for example, can be implemented at either runtime or load time.
  - **Configuration files** are intended to set parameters at startup.
  - **Polymorphism** allows late binding of method calls.
  - **Dependency injection** allow late binding of dependencies.