Inoculation Pattern in Change Management

Gautam Hegde/Koushik H

gautam@backend.co.in/koushik@backend.co.in

Backend Bangalore Pvt. Ltd. #587, KPC Layout, Bangalore - 560035

Abstract

The management of change becomes very challenging for the implementers and users in the live environment of a software implementation. The change necessitated due to increased visibility into the user requirements or due to underlying technology changes have to be brought about in real time without bringing down the system. The inoculation pattern suggests bringing about the essential change in a gradual and staged manner keeping the quantum of change well below the user's rejection threshold. The process comprises of –

1. Identifying, Ordering and Selecting functional modules in terms of importance and usage from a user perspective
2. Injecting the change in selected modules;
3. Allowing sufficient familiarity in these modules before the changes are introduced to the rest, until all of the modules are covered and the whole change is implemented.

Thumbnail

If

- Changes are required due to additions/alteration in user requirements;
- Due to change in underlying technology

Then

- Identify modules in terms of importance to user and usage
- Injecting change in selected modules
- Allow for incubation of familiarity with the introduced change, before inoculating rest of the modules

Problem

Changes are inevitable in software process and happen due to alteration and additions to user requirement. Changes can also happen due to change in underlying technology. Incorporating these changes in live environment can be extremely challenging. Implementers do not have luxury of downtime or long drawn user training schedule. The implementation of changes and user training has to

be done on the fly. User resistances to changes have to be overcome smoothly so that their rejection threshold is not breached. Business continuity is of paramount importance since discontinued services mean risking loss of business. So the change management program has to evolve a method for introducing the changes without risking business continuity.

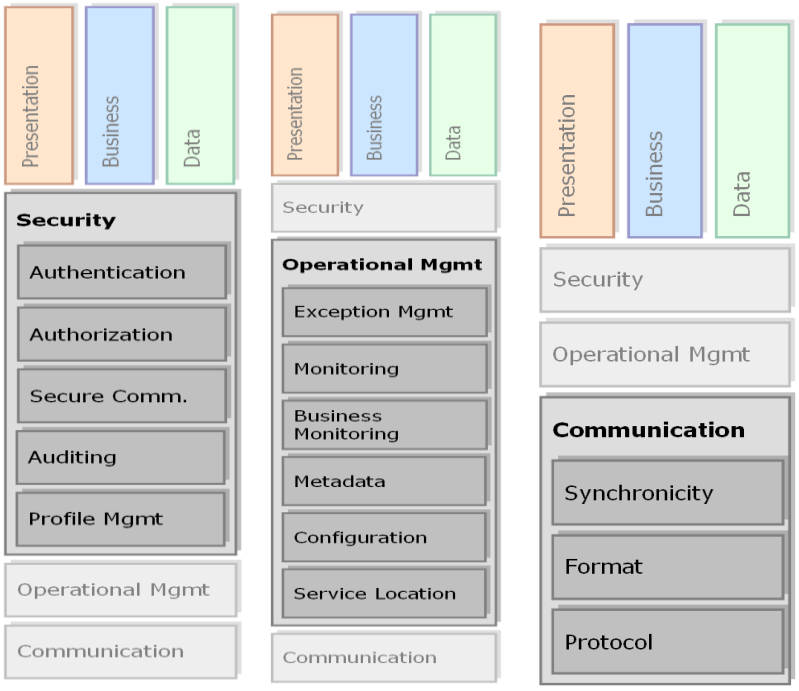
Forces

- Scope creep: documented scope in software development process is usually like a tip of ice-berg with a large portion of work hidden at the requirement capture stage.
- Increase in visibility of additional requirements during the course of implementation.

Solutions

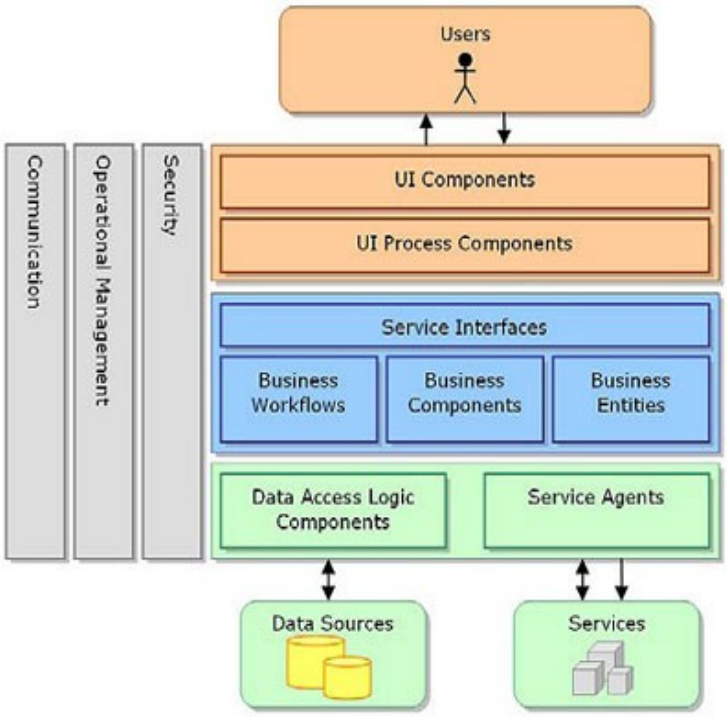
Software Architecture has to cater to inevitable future changes. While the visibilities to the changes are limited, an n-tier approach may help by separating the functionalities in different layers.

Software architecture could be visualized at the logical level as abstract layers comprising of specific set of functional components interacting with each other to deliver specific business functions.



Functional modules may be grouped logically into following layers or logical block:

- Database/Storage (DB):
  This is the layer the persistent data is stored as tables or files.
- Data Access Layer (DAL):
  In this layer the methods for accessing Database is defined. As methods of accessing different DBs are different, this layer hides/abstracts the details from top layers.
- Business Logic Layer (BLL):
  In this layer addresses the functions of business where business logic decides the data to be collected and directed into DB through DAL. This layer also provides interfaces to top layers as well as feature interaction among layers.
- Transport Service Layer (TSL):
  This layer comprises of communication and security protocols defining means of transportation of information functioning as an interface between Business Logic Layer and User Interface Layer.
- User Interface (UI):
  This is the layer visible to user through which they make use of the business logic to achieve the objective of software application. The users interact with application server side software through this layer using client side software like web browser or forms.



Each layer interacts with its adjacent layer [$Layer_X$] [upper ($Layer_{X+1}$) or lower ($Layer_{X-1}$)]

To implement a change in a software system, multiple layers may be affected, which may break existing functionalities. Change may be in

A. UI layer or
B. Underlying layers in that module with corresponding UI change or
C. Only in specific layer(s) but impacting user experience without affecting UI

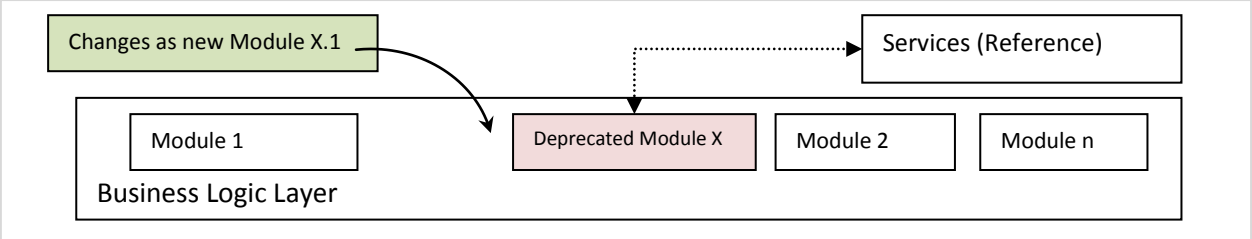
Changes in the UI and/or changes in the user experiences increase the chance of user discomfort.

One approach is to surgically replace the whole functional module with a new enhanced module. In such cases probability of user discomfort could be high resulting in larger outlays of time and resources for re-training. To avoid this, an approach can be taken which gradually changes the system instead of doing a drastic change. This could be particularly effective where there are eye-visible changes.

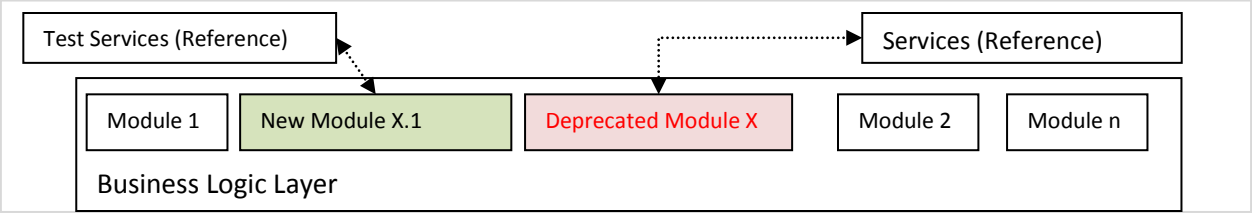
The gradual change approach helps to maintain the continuity of the services. Same approach may be applied to each functional module and its dependant module.

Example: Injecting a change in Business Logic Layer

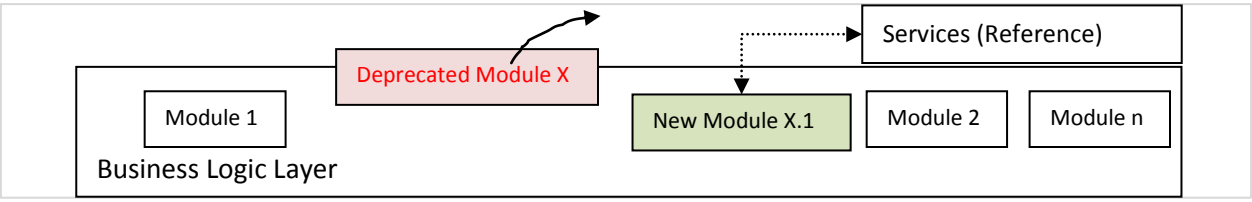
Stage 1: Create change as new entity



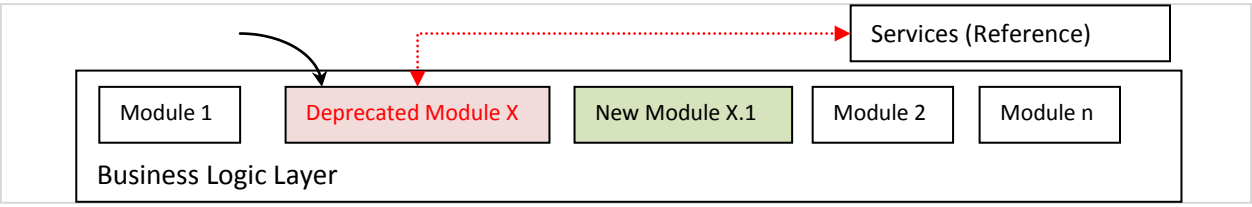
Stage 2: New block (Changed) and existing block co-exist so that if there is a problem there can be easy roll back.



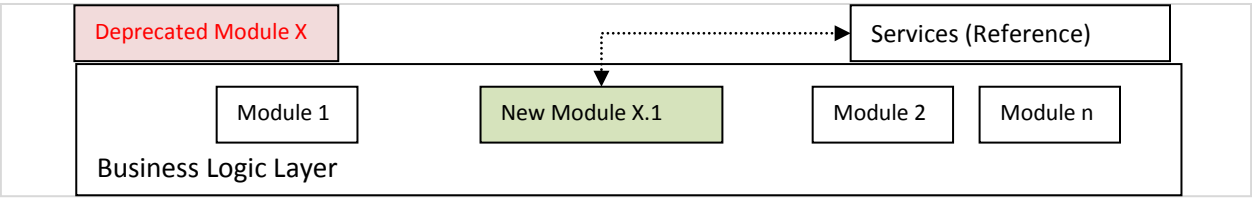
Stage 3: Deprecated block removed after successful testing of new block. All references to the deprecated block from rest of the software also removed.



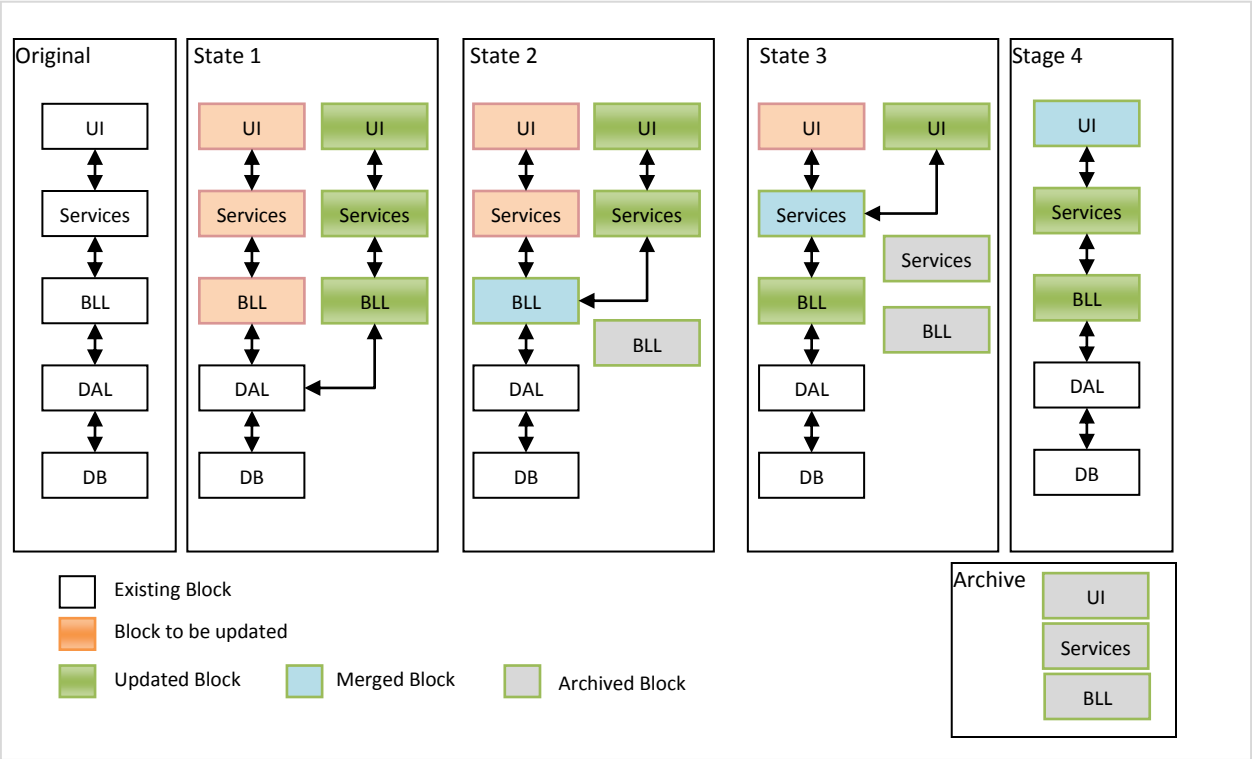
In case of any necessity of Rollback the Deprecated Block and it references can be brought to its original positions.



Stage 4: Change completed. Deprecated block is archived.



Example: If change affects a single functional block



In this case all the dependant layers are duplicated and merged at each stage with consequent changes in dependant interfaces. Co-existing old block and new block helps to keep backward compatibility and rollback if there is any necessity. After each dependant block is merged, old block's content can be removed/archived.

Resulting Context

Enhanced functionalities achieved without disruption of any business services and user re-trained with minimal time and effort.

Conclusion

Inoculation pattern is most effective in cases of eye-visible change management. There could be some limitations to this pattern in cases where changes impact user experience without consequent changes to the UI.

Acknowledgement

We would like to acknowledge Prof. K V Dinesha of International Institute of Information Technology – Bangalore for enlightening us on PLoP and shepherding us in the preparation of this paper by holding the preparatory workshop. We also like to thank Mr. Éric Platon for shepherding this paper.