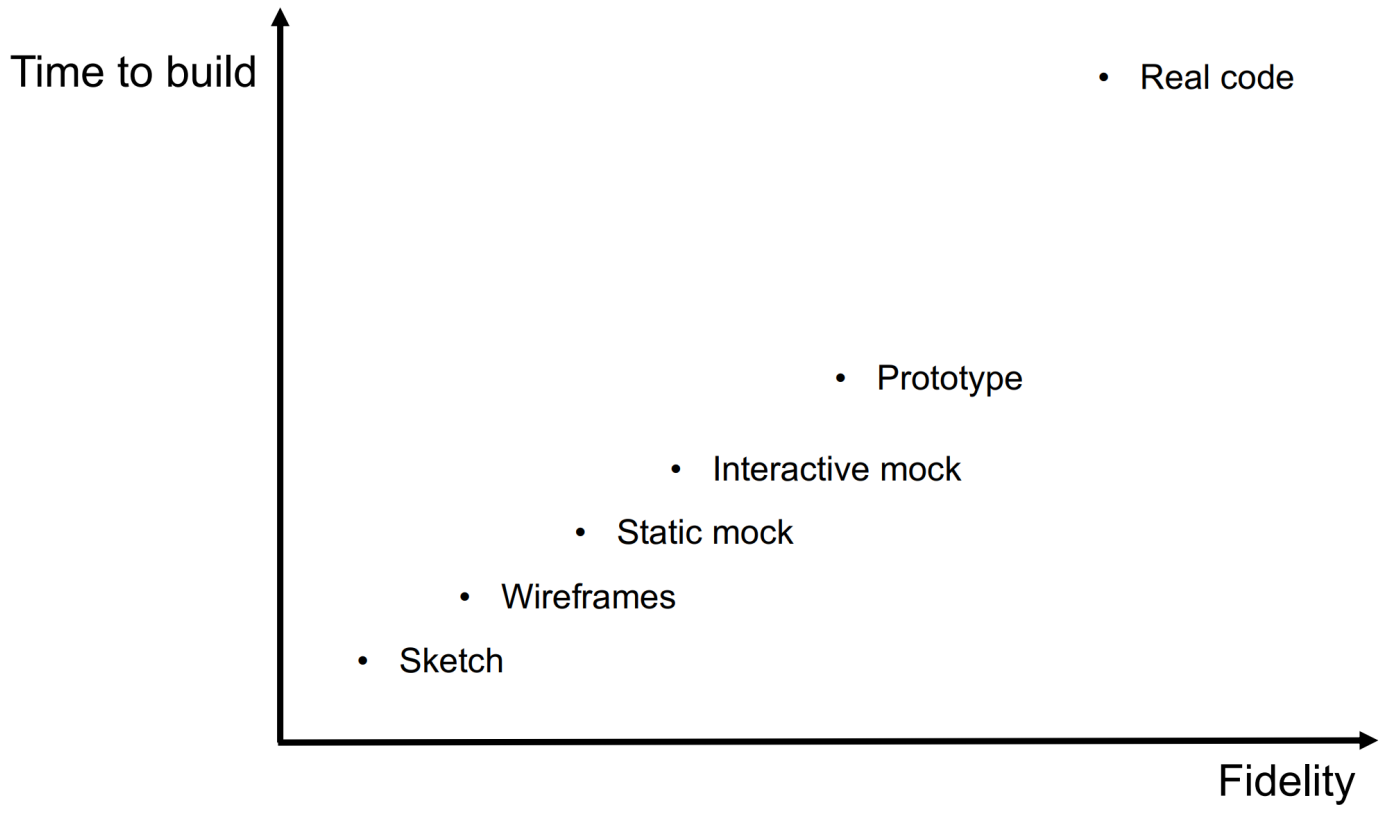
COMP23412 Software Engineering 2 (30)

Model-View-Controller (MVC) is a way of developing software.

Models represent knowledge. A model could be a single object (rather uninteresting), or it could be some structure of objects. There should be a one-to-one correspondence between the model and its parts on the one hand, and the represented world as perceived by the owner of the model on the other hand. The nodes of a model should therefore represent an identifiable part of the problem. The nodes of a model should all be on the same problem level, it is confusing and considered bad form to mix problem-oriented nodes (e.g. calendar appointments) with implementation details (e.g. paragraphs).

View is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a *presentation filter*. A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages. All these questions and messages have to be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents. (It may, for example, ask for the model's identifier and expect an instance of Text, it may not assume that the model is of class Text.)

Controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on .to one or more of the views. A controller should never supplement the views, it should for example never connect the views of nodes by drawing arrows between them. Conversely, a view should never know about user input, such as mouse operations and keystrokes. It should always be possible to write a method in a controller that sends messages to views which exactly reproduce any sequence of user commands.



There are many guidelines sets. The one examinable in Software Engineering 2 is Ben Shneiderman's 8 golden rules:

1. Strive for consistency. Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent colour, layout, capitalisation, fonts, and so on, should be employed throughout. Exceptions, such as required confirmation of the delete command or no echoing of passwords, should be comprehensible and limited in number.

2. Seek universal usability. Recognize the needs of diverse users and design for plasticity, facilitating transformation of content. Novice to expert differences, age ranges, disabilities, international variations, and technological diversity each enrich the spectrum of requirements that guides design. Adding features for novices, such as explanations, and features for experts, such as shortcuts and faster pacing, enriches the interface design and improves perceived quality.

3. Offer informative feedback. For every user action, there should be an interface feedback. For frequent and minor actions, the response can be modest, whereas for infrequent and major actions, the response should be more substantial. Visual presentation of the objects of interest provides a convenient environment for showing changes explicitly.

4. Design dialogs to yield closure. Sequences of actions should be organised into groups with a beginning, middle, and end. Informative feedback at the completion of a group of actions gives users the satisfaction of accomplishment, a sense of relief, a signal to drop contingency plans from their minds, and an indicator to prepare for the next group of actions. For example, e-commerce websites move users from selecting products to the checkout, ending with a clear confirmation page that completes the transaction.

5. Prevent errors. As much as possible, design the interface so that users cannot make serious errors; for example, grey out menu items that are not appropriate and do not allow alphabetic characters in numeric entry fields. If users make an error, the interface should offer simple, constructive, and specific instructions for recovery. For example, users should not have to retype an entire name-address form if they enter an invalid zip code but rather should be guided to repair only the faulty part. Erroneous actions should leave the interface state unchanged, or the interface should give instructions about restoring the state.

6. Permit easy reversal of actions. As much as possible, actions should be reversible. This feature relieves anxiety, since users know that errors can be undone, and encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data-entry task, or a complete group of actions, such as entry of a name-address block.

7. Keep users in control. Experienced users strongly desire the sense that they are in charge of the interface and that the interface responds to their actions. They don't want surprises or changes in familiar behaviour, and they are annoyed by tedious data-entry sequences, difficulty in obtaining necessary information, and inability to produce their desired result.

8. Reduce short-term memory load. Humans' limited capacity for information processing in short-term memory (the rule of thumb is that people can remember "seven plus or minus two chunks" of information) requires that designers avoid interfaces in which users must remember information from one display and then use that information on another display. It means that cell phones should not require re-entry of phone numbers, website locations should remain visible, and lengthy forms should be compacted to fit a single display.

These underlying principles must be interpreted, refined, and extended for each environment. They have their limitations, but they provide a good starting point for mobile, desktop, and web designers. The principles presented in the ensuing sections focus on increasing users' productivity by providing

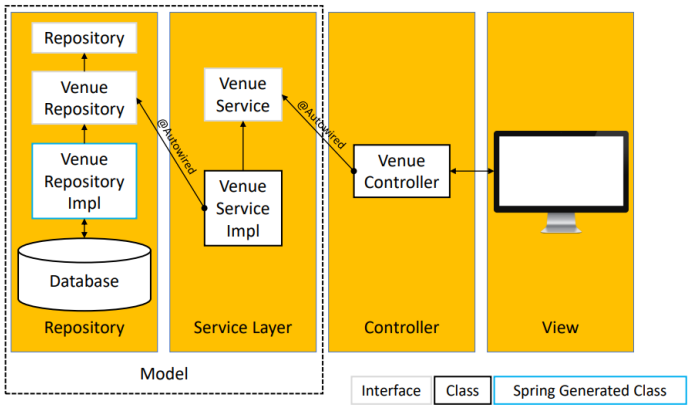
simplified data-entry procedures, comprehensible displays, and rapid informative feedback to increase feelings of competence, mastery, and control over the system.

Dark Pattern is a type of user interface that is designed to trick or deceive users into doing something they do not want to do.

Plain Old Java Object (POJO) is an ordinary Java object, not bound by any special restriction. It should NOT extend prespecified classes, implement prespecified interfaces, or contain prespecified annotations.

Create, Read, Update, and Delete (CRUD) are four basic operations of persistent storage. The default implementation of CRUD Repository provides: `count()`, `findOne(long id)`, `findAll()`, `save(Item item)`, `delete(long id)`, and more.

Spring's Data Access Objects



Secure Authentication requires hashing with salt at client, and secured hashed result in backend store. MD5 is now considered insecure, SHA1 is not recommended. SHA2/3, b/scrypt, Lyra2, and Argon 2 is believed to be secure.

Authorisation is the process of specifying access rights/privileges to resources. It protects against privilege escalation attacks. Role-based authorisation is commonly used.

Injection Attack happens when an attacker injects untrusted input which gets processed as part of a command or query. Common types include *Cross-Site Request Forgery (CSRF)*, *Cross-site scripting (XSS)*, and *SQL injection*. These could be prevented by validating user input, and the primary way of prevention is to never trust user input (or, client integrity).

Hoepman's Eight Privacy Design Strategies are: ***Minimise*** (data collection; select, exclude, strip, destroy), ***Separate*** (the processing to hard to combine; isolate, distribute), ***Abstract*** (the detail during processing; summarise, group, perturb), ***Hide*** (make it not linkable or unobservable; restrict, obfuscate, dissociate, mix), ***Inform*** (users about the processing; supply, explain, notify), ***Control*** (provide user adequate control over processing; consent, choose, update, retract), ***Enforce*** (commit and do process privacy-friendly; create, maintain, uphold), ***Demonstrate*** (the practices to users; record, audit, report).

Dummy is passed around but never used.
Fake generally works as expected, but has some shortcut unsuitable for full production.
Stub provides a canned answer to a particular invocation.
Mock has pre-programmed expectations of how it will be called, and what will happen internally when it is called.

Software as Service (SaaS) = API + resources + interface.

Security-by-Design Approach is completed in the order of: Identify assets; Perform threat modelling; Choose mitigation techniques; Design and test.