LEAN Model Based Experimentation on UI Prototypes

Using Task Based Usability Testing

Rakshit Bhat

Supervisor: Dr. Enes Yigitbas
Prof. Dr. Gregor Engels

Advisor: Mr. Sebastian Gottschalk

Department of Computer Science
Universität Paderborn / Paderborn University

Master of Science

January 2023

I would like to *dedicate* `this` **thesis** to ...

Official Declaration
Eidesstattliche Erklärung

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or a substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Date	Rakshit Bhat

Acknowledgements

And I would **like** to acknowledge . . .

‘’

Abstract

The user interface (UI) layer is one of the essential aspects of software applications since it associates end-users with the functionality. For interactive applications, the usability and convenience of the UI are essential factors for achieving user acceptability. Therefore, the software is successful from the end user's perspective if it facilitates good interaction between users and the system.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

This chapter motivates the readers about the topic (see section 1.1), explains the problems faced by the companies during software development (see section 1.2), our research approach (see section 1.3), and finally, our solution approach (see section 1.4).

1.1 Motivation

Over the last decade, software development had a tremendous impact with increasing customer demand and requirements [2], which increases product complexity and ambiguity, significantly impacting software development. Therefore, the developers have come up with different techniques to meet this requirement criteria. Early user feedback from potential customers in the industry is crucial for creating successful software products because of the growing market uncertainties, and consumers' desire to receive integrated solutions to their issues rather than unique software developments [3]. With the increasing complexity of products, it becomes challenging to determine user requirements making it more difficult for developers to assess their opinions. As a result, the developers of these products are biased toward some requirements and can ignore what the user wants. So, the developers must detect the user's needs and requirements to reduce these risks early. Giving users a "partially functioning" system is the most excellent method to determine their requirements and suggestions [4]. This ensures that the developers with high uncertainties in the early product development phase can improve the product by testing the underlying assumptions [5]. Developers can use this feedback to validate the most critical assumptions about the software product. This validation can decide whether to add, remove or update a feature [6]. This process of determining the best fit for the product through user feedback is called experimentation. There has been an increase in interest in the types of experimentation that can take place in product development. Software products have shown the benefits

of conducting experiments in many use cases with incremental product improvement [7]. In experimentation, the product designers design different UI variants (e.g., buttons with different colors), and the developer integrates these variants and assigns them to a distinct group of users. As per some evaluation criteria (e.g., more clicks on the button), the variant with better results is deployed for the entire set of users. So, an experiment can be valuable when it improves the software products. Hence, for experiments to be successful, they should offer one or more solutions that will benefit users.

1.2 Problem Statement

The motivation section shows some gaps in software development between the developers and the designers. This section explains the problems and determines their research and solution approach.

Problem 1: Product designers create many UI prototypes, and the developers implement them. To determine the best variant, the developers create experiments with the users [6]. This concrete implementation of designs uses a lot of resources and time for the developers. Therefore, the product designers need to be integrated into the development process so that they would be able to create experiments independent of the developers.

Problem 2: When the product designers develop the prototypes, testing them with many users is difficult as the product is still not developed. Therefore, it is not easy to conclude a "winner" variant with a small amount of data as it is statistically difficult to prove one of the variants outperforms the others [8]. Therefore, it is necessary to develop an idea that the designers can use to determine the best prototype or variant with a small group of users.

Problem 3: Most often, the software application collects data from the experiments. Some data is used in qualitative analysis, while others are in quantitative analysis. Many companies fail to reap the benefits of using both qualitative and quantitative analysis. Similarly, not all the data is used in the analysis phase reducing the software applications to improve based on customer feedback [9]. Therefore, finding a solution that combines qualitative and quantitative data analysis is necessary.

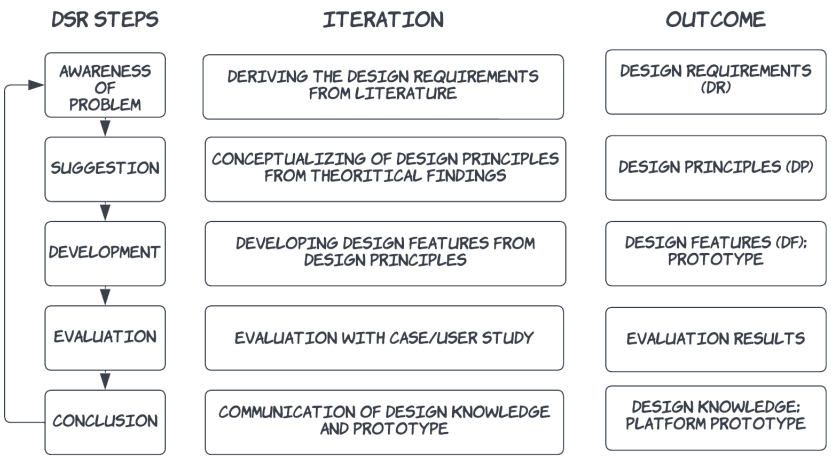


Fig. 1.1 Design Science Research Cycle [1]

1.3 Research Approach

The process of creating experiments and testing their variants is usually not systematically arranged, creating anomalies, and leading to unsuccessful experiments. Therefore, this section identifies the research question (RQ) and defines an approach to answer the question.

RQ: *How to develop a platform suitable for product designers to conduct experiments on UI prototypes, increasing its usability and, simultaneously, independent of developers?*

Prescriptive knowledge about the design of artifacts, such as software techniques, models, or concepts, is what design science research (DSR) aims to provide. Due to this design knowledge, future projects can design artifacts methodically and scientifically with the aid of study and practice [1]. Therefore, we will conduct a design science research (DSR) study to answer our research question and obtain abstract design knowledge and an implementation tool. From the abstracted knowledge, we will obtain some Design Principles (DPs) defined for the whole process of experimentation [1]. In this design, the product designers will iteratively validate their prototypes with the users (or the crowds). Here, DPs capture and codify that knowledge by focussing on the implementer, the aim, the user, the context, the mechanism, the enactors, and the rationale [10]. The DPs explain the design information that develops features for software applications. We propose to use the variation of the cycle of Kuechler and Vaishnavi [1] consisting of five iteratively conducted steps (see figure 1.1). As a result, this design and application may provide design-focused information that adds

to the DSR knowledge corpus [11]. Every element within a DSR project is built upon and systematically analyzed to add to the overall DSR knowledge corpus. Therefore through the use of DSR, a group of issues is resolved by concentrating on a single issue and abstracting the consequences of the resolution.

Design Requirements (DRs): In design science research, abstracted *Design Requirements (DRs)* refer to the general, high-level requirements that a design must meet to succeed. These requirements are typically derived from the research question identified as relevant to the problem. Abstracted design requirements provide a broad framework for the design process and help to ensure that the design solution addresses the key issues and challenges identified in the research problem. They can also help guide the evaluation of the design solution and ensure that the solution is grounded in the design principles identified as necessary.

Design Principles (DPs): In design science research, concrete *Design Principles (DPs)* refer to specific, detailed guidelines that a design must follow to be successful. These principles are derived from the abstract design requirements identified as relevant to the research problem and provide more specific guidance on how the design should be implemented. DPs can also be defined as the codification of our knowledge during the design study while identifying the DRs. Concrete DPs are essential in DSR as they help ensure that the design solution meets the needs and goals of the research problem and is grounded in the design principles we identified as necessary [12]. More details about the DRs and DPs can be found in section 3.1.

1.4 Solution Approach

To solve the problems mentioned above, the designers should be able to create UI prototypes and experiments on their own on a set of users. Since we do not have a large set of users for testing the prototypes, we use supervised task-based usability testing [13]. The fundamental principle of task-based usability testing is to have the users attempt to use the prototypes to do certain activities or tasks (e.g., Locate a movie M1) and get feedback (e.g., the time required for the task to be completed by the user). We propose to use `Low-code` or `No-Code` approach to achieve this. This approach helps to have a UI for the designers to understand, develop, and create experiments and tasks with the software prototypes [14]. So, the designers would be able to create the UI prototypes and their variants, assign them to the users in an experiment, get feedback from the users and decide on the best prototype. At the same time, the low-code has become more accessible for Model-driven development [15]. Therefore, we plan to

create models for the UI prototypes and have the feasibility for creating experiments and tasks. Because of using the models, it is easier to store the prototypes in the database and conduct experiments with the users.

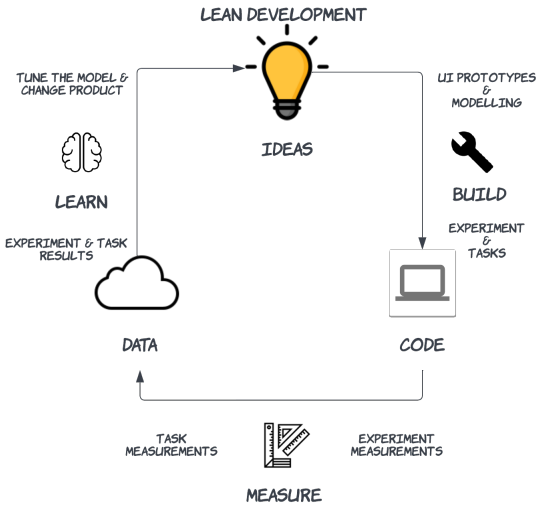


Fig. 1.2 LEAN Development technique

In our solution, we use the LEAN development technique (see figure 1.2) for development as it is used to develop customers friendly products [16]. Using LEAN, the company creates a Minimum Viable Product (MVP) throughout development, tests it with potential customers, and leverages their input to make incremental changes. While this technique can be used for every product, there are also approaches specific to software products. LEAN development technique can be divided into a Build, Measure, and Learn cycle. In the `(1) Build` phase, we plan to create the `UI Prototypes`, `Models`, `Experiments`, and `Tasks` for the users. In the `(2) Measure` phase, we plan to assign the Experiments and Tasks to the users and measure the `Task and the Experiment measurements` and perform some analysis on the data received. And finally, in the `(3) Learn` phase, we display the `Analyses results`, `Tune` our models to decide the better variant among the others, and `Modify` the prototype. As per the figure 1.2, we complete one cycle of iteration and start a new one with the updated prototype.

Chapter 2

Foundations

The foundation of this research is built upon the existing knowledge and theories developed in the field of Software Engineering. This chapter provides an overview of the relevant literature and identifies the key concepts and theories necessary for the research. Firstly in section 2.1, we will discuss *UI Prototyping* explaining the creation of a User Interface (UI) simulation and testing and refining the design ideas and user requirements. Then, in section 2.2, we explain *Low/No-code methods* and citizen development of the software products. Then, in section 2.3, we describe the *Model-Based Software Engineering (MBSE)* approaches and meta-models. In section 2.4, we clarify *Task-based Usability Testing* for reviewing and refining the product. Later in section 2.5, we define the concept of *Experimentation* and *A/B testing*. Finally, in section 2.6, we explain the *LEAN software development process*, explaining the Build, Measure and Learn phases.

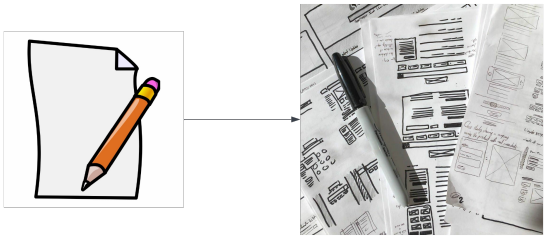
2.1 UI Prototyping

UI prototyping creates a simplified UI version to test and iterate on design ideas before building the final product. UI prototyping is a valuable tool for designers, allowing them to quickly and easily test and refine UI designs [17]. It also helps gather feedback from users and stakeholders [18] and identify any usability issues before investing significant time and resources into the final product. From paper to Hypertext Markup Language (HTML) code, everything could be a prototype, including various techniques like `Paper Prototypes`[1],

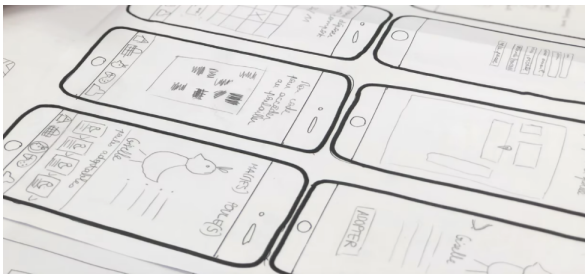
[1] **Paper prototyping:** It is a method of creating a preliminary version of a UI using paper and pen.

`Wireframes`[2], `Mockups`[3] and `Interactive Prototypes`[4]. Moreover, UI Prototypes can be classified as *High-Fidelity* and *Low-Fidelity* prototypes. The comparison by Jim Rudd et al. [19] on high and low-fidelity prototyping, based on its advantages and disadvantages, is explained further.

Low-Fidelity Prototypes: *Low-fidelity* prototypes (see figure 2.1) are prototypes usually with limited functions and little interaction prototyping effort. They mainly focus on explaining concepts, design alternatives, and screen layouts. Storyboard presentations, cards, proof of concept prototypes, paper prototypes, etc., come under this category. E.g., Paper-based prototypes are simple, low-fidelity mockups that can be quickly and cheaply created using paper and pencil (see figure 2.1a). In this case, the UI designers use simple text, lines, and forms to hand-draw concepts. Instead of aesthetics, the focus is on speed and creative ideas. To simulate user flows (as shown by figure 2.1b), designers lay paper screens on the floor, table, or pinned to a board. Therefore, these prototypes emphasize communicating, educating, and informing rather than training, testing, and codification [20]. The advantages of low-fidelity prototypes are rapid development, lower development cost, addressing issues, and usefulness for a proof-of-concept [19]. Similarly, the disadvantages include limited error checking, difficulty with usability testing, navigation, flow limitation, etc.



(a) Sketches and whiteboard for prototyping



(b) Example on how the paper-based prototypes are developed [21]

Fig. 2.1 Low fidelity prototyping

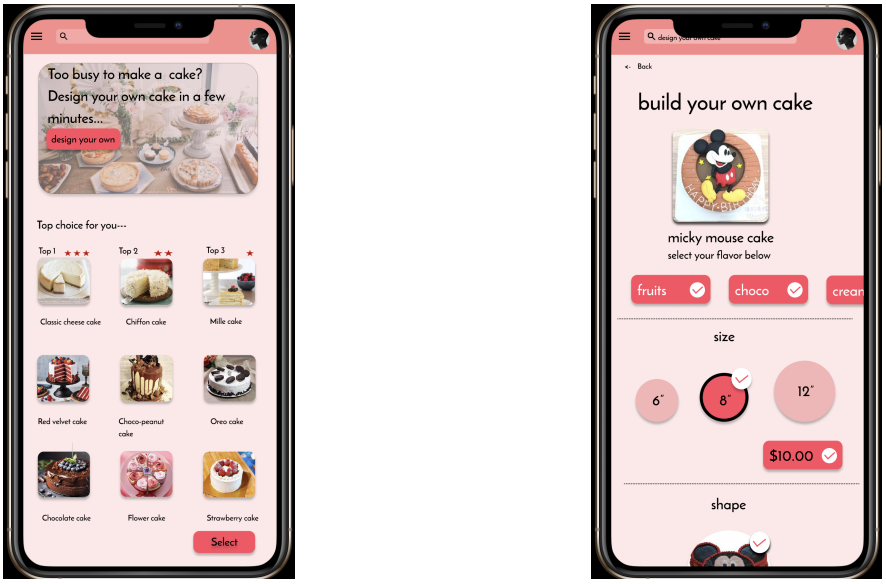
High-Fidelity Prototypes: Contrary to low-fidelity prototypes, *High-fidelity* prototypes (see figure 2.2) have full functionality and focus on flow, and the user models of the system

[2]**Wireframe:** A wireframe is an essential visual representation of the layout and structure of a product, showing the placement of text, images, and buttons.

[3]**Mockup:** A mockup is a more detailed version of a wireframe, typically created in color and with more realistic graphics and images.

[4]**Interactive Prototype:** An interactive prototype is a fully functional representation of a product, design, or feature.
Lookup **Appendix B** for a more detailed explanation.

[22]. These prototypes are detailed, interactive mockups of a UI designed to mimic the look and feel of the final product closely. High-fidelity prototypes are often created using interactive prototyping tools, which allow designers to create realistic, interactive mockups of UI designs [23]. These tools typically include a library of UI elements and templates and the ability to create interactive transitions and animations. The users can operate these prototypes, and the developers can collect information from the users through measurements. Some advantages of high-fidelity prototypes are that they are user-driven, used for navigation and tests, and can also be served as a marketing tool for attracting potential customers [19]. Therefore, these prototypes are typically more realistic and interactive than those created using other tools but may require more time and resources to develop and maintain. At the same time, high-fidelity prototypes allow designers and developers to test and refine complex UI concepts and interactions, ensuring that the final product is user-friendly and practical. Overall, low-fidelity prototypes and high-fidelity prototypes are both valuable tools in the design process, and the choice between the two will depend on the specific needs and goals of the project.



(a) Prototyping page of a cake company (b) Prototyping page to customize a cake

Fig. 2.2 High Fidelity prototype: Model-based UI Prototyping [24]

UI prototyping is an evaluation and testing technique according to User-Centered Design (UCD) methodology since the 1990s [25]. The evaluation of prototypes by users gives crucial feedback in iterative approaches for Information Technology (IT) project management, especially agile methodologies [26]. Therefore, to build an exemplary UI, a company can use this approach: develop a preliminary version of the UI, test it with people, and make as many

revisions as possible (without building the actual software) [17]. Figure 2.3 shows a cycle representing the iterative development with prototypes. The designers start the process by developing the UI prototypes, which the stakeholders (e.g., customers and product managers) review. The UI prototypes are refined from the feedback received, reiterating the cycle. Therefore, designing UI prototypes enables designers and stakeholders to communicate more effectively. Similarly, an interactive prototype helps visualize design concepts and

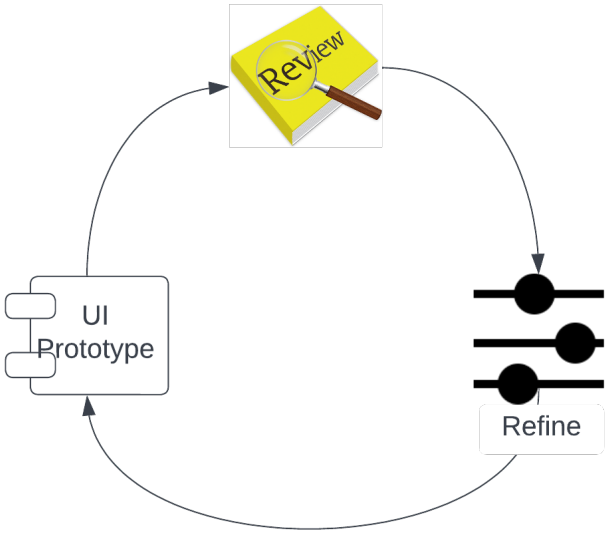


Fig. 2.3 Prototyping steps for an iterative development

communicate new requirements and expectations about a prospective system. Iterative design requires multiple updates to the design's execution.

Since developing and updating the entire software system is complex and expensive, prototyping is a crucial technique [27]. Simultaneously, software prototypes might exclude many requirements, making the software more accessible, smaller, and less expensive to construct and change [27]. Thus, the main difference between a prototype and a software application is that in a prototype, the displays are designed images with no additional capabilities to display the design and flow. Moreover, usability testing to validate user requirements and prototype functionality is part of the evaluation process for UI prototypes [28]. Thus by using prototyping, there is usually more contact between the designers and users, resulting in fewer usability flaws and corrections at the end of development. And finally, these mockups are then converted into actual UI elements in a software application, and a flow is available.

Overall, UI prototyping is an essential part of the design process. It allows designers and developers to quickly test and refine UI concepts, gather feedback from stakeholders and users, and identify any usability issues before investing significant time and resources into building the final product. Whether using low-fidelity prototypes to test basic layout and navigation concepts or high-fidelity prototypes to test complicated UI concepts and interactions, prototyping can help ensure that the final product is user-friendly and effective. By iterating and refining their prototypes throughout the design process, designers can create better, more intuitive products that meet the needs of their users.

2.2 Low Code / No Code Development Platform

Low code is a software development method that uses less human coding to enable users to construct and manage programs efficiently rather than writing extensive amounts of code [29]. It is a technique used by developers to help non-developers design and develop software applications using a *Graphical User Interface* (GUI) supported by a *Low-Code Development Platform (LCDP)*. An LCDP allows developers to create, deploy, and manage applications quickly and easily using high-level programming languages and techniques such as model-driven and metadata-based programming [30]. It simplifies building and deploying applications using declarative programming abstractions and one-step deployments. LCDP provides pre-built components, templates, and other resources that companies can quickly assemble to create functional applications. These platforms are designed to accelerate the development process and enable companies to quickly build and deploy custom software solutions. Similarly, there is another technique called *No-code development* supported by the *No-Code Development Platform (NCDP)* [31]. Unlike low code, no-code platforms require no programming skills because they offer drag-and-drop techniques for building the apps. The non-developers can easily pick up the components which fit the UI and drag and drop them to the screen and finally create an entire application using this technique. Using the user interface and ready-made automatic tools on these application development platforms, it is feasible to create apps relatively quickly. Due to its simplicity, flexibility, and low cost, companies have started using this platform to meet the high demands of software development and digitalization [29]. Additionally, with its self-configurable components, it lowers the expenses associated with initial installation, training, distribution, and maintenance [32].

High-level architecture of LCDP: As shown in figure 2.4, an LCDP is divided into small modules independent of the other components. Bock et al. [30] have classified these modules according to the three main approaches to systems development: *the static perspective*, *the functional perspective*, and *the dynamic view*.

Static Perspective: LCDPs typically allow data to be stored either in an internal Database Management System (DBMS) or in external systems (see the first part of figure 2.4). Many of these features/components commonly found in LCDPs fall under the *static perspective*. Similarly, most LCDPs include a component for defining Data Structure (DS), usually provided as a conceptual modeling tool that uses a classical Data Modeling Language (DML) such as the Entity-Relationship Model or a Domain Specific Language (DSL). Some LCDPs allow DS to be defined only through UI-based dialogs or lists. For example, you are allowed

to upload a Comma Separated Value (CSV) file that contains data that is persisted into the Database (DB) as per the columns in the file. A common feature of LCDPs is the ability to access external data sources using various Application Programming Interfaces (API).

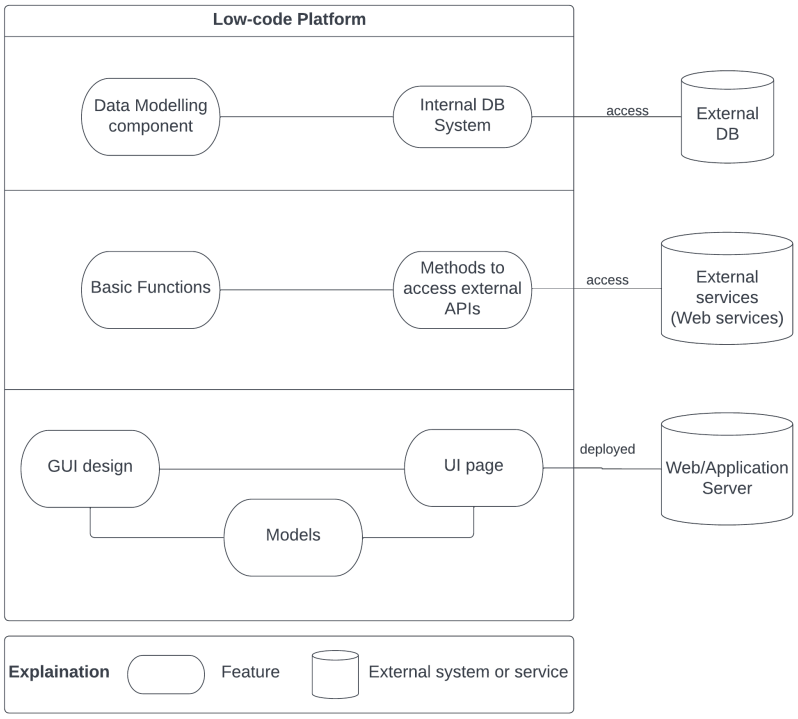


Fig. 2.4 High-level architecture of Low code Development platform

Functional Perspective: LCDPs also offer basic functional specifications (see the second part of figure 2.4). These typically include simple expression languages for decision rules and dialog-based methods for specifying program flow conditions. Each solution consists of a library of generic standard operations, such as mathematical functions. For example, LCDPs will enable the use of traditional approaches such as web services and Representational State Transfer (REST) services, and many modules provide support for a wide range of APIs from individual providers, such as Google APIs, and social media APIs [30].

Dynamic Perspective: The LCDP also includes a Graphical User Interface (GUI) designer module (see the third part of 2.4), which allows for the development of GUIs and their integration with other implementation elements. The GUI designers module specifies pre-defined widgets, although the range varies. It is generally easy to link GUIs and DSs in most of these platforms, as it is optional to implement the Model-View-Controller (MVC) pattern

manually. In addition, many of these platforms offer specific support for adapting the GUIs to different target environments, such as desktop browsers, tablets, and smartphones [33]. Another common feature is the inclusion of a component for defining roles and user rights, which is usually part of the platform's governing architecture and is deployed along with the custom application [29].

*Main Steps Of **Low-Code** App Development* as per [34]:

Building: In this step, the platform gives you the freedom to alter the provided code and add hand-written custom code to it to specify more complex features in the app as you create the app step-by-step using visual editors and drag-and-drop interfaces. Modules, components, and chart-builders are already incorporated into low-code applications. Charts may be used to display data from modules, while modules are used to specify the type of data that will be stored in the app. Components and pages provide the type of user experience the app will have. These platforms also have a provision for automating repetitive tasks in the app.

Testing: Testing a software application is an integral part of the development cycle. However, the low-code development platform decreases the requirement for testing. Pre-build modules and components on low-code platforms are created with a certain level of application security. The developers of the low-code platform constantly monitor these modules, and they have previously gone through several unit tests. But, in a low code platform, testing can be performed in several ways [35]:
- *Automated testing:* Some low-code platforms have built-in support for automated testing, which allows developers to create and run tests that validate the functionality of their applications.
- *Manual testing:* Even with automated testing, it is necessary to perform manual testing to ensure that an application functions correctly. In a low-code platform, manual testing can be achieved by developers or by dedicated testers who are responsible for verifying the functionality of the application.
- *User Acceptance testing (UAT):* In some cases, it may be necessary to involve end users in the testing process. This can be done through user acceptance testing, which consists in having end users test the application and provide feedback on its functionality and usability.

Overall, testing in a LCDP involves a combination of automated and manual testing, as well as performance and acceptance testing [35], to ensure that the application is functioning correctly and meets the needs of end users.

Deploying: In this step, the application is deployed across apps and to the final users. In LCDP, the packages for installation, configurations, and application setup are included. In terms of deployment, most of the considered solutions offer advanced support, although the specific forms vary. Some systems require the low-code platform environment to be installed on a web server to deploy individual applications. In contrast, others allow the developed solutions to be deployed as self-contained applications on various devices and machines. This means that the LCDP gives freedom to the users in deploying the applications for the customers with just one click.
Docker[5] is a widely used open software platform that allows applications to be deployed as containers. It supports libraries, system tools, code, and various runtimes, making it possible to build, test, deploy, and scale applications across multiple environments. In an LCDP, docker can be used to securely deploy frameworks, services, and platforms in separate containers that communicate with each other using protocols such as Message Queue Telemetry Transport (MQTT)[6] and REST[7]. Additionally, a variety of options are provided for developers with little programming experience, those with coding expertise and seasoned programmers who wish to expand the functionality of the current design [29].

Overall, low-code/no-code development platforms are valuable for organizations looking to build and deploy custom applications quickly and efficiently that can help organizations of all sizes and industries bring their ideas to life.

[5]Docker: https://www.docker.com/
[6]MQTT: The Standard for IoT Messaging https://mqtt.org/
[7]REST API (also known as RESTful API) https://www.redhat.com/en/topics/api/what-is-a-rest-api

2.3 Model-based Software Engineering

MBSE refers to maintaining and developing software while reusing existing code. Similarly, Model Driven Software Engineering (MDSE) is the term used to cover various techniques for creating software using codified models. At the same time, for creating models, the Model Object Facility (MOF) has defined four levels separating the reality, models, meta-models, and meta-meta-models (as shown in the figure 2.5).

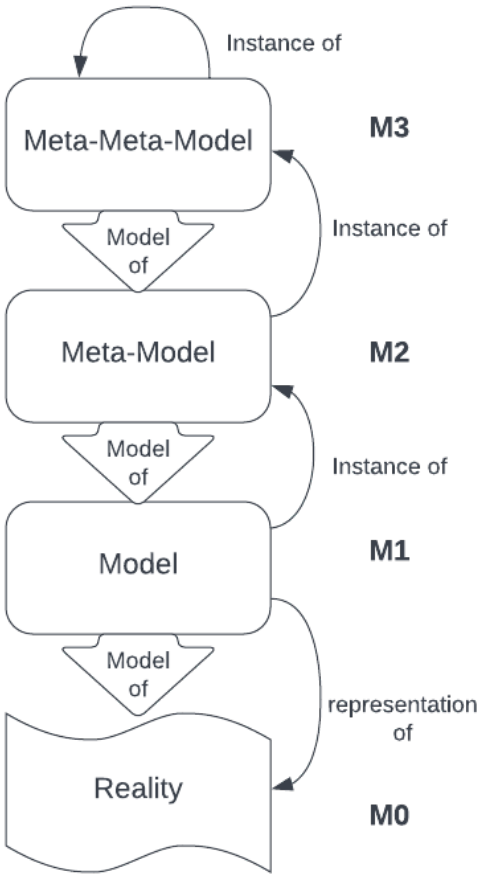


Fig. 2.5 Model Object Facility (MOF) levels

MOF is a standard that defines a meta-model for modeling information. It provides a common framework for creating and using models representing a system or part of a technique used to analyze design or document the procedure. MOF is part of the Object Management Group (OMG) [36] standards and is used in several different contexts, including software engineering, business process modeling, and enterprise architecture. It is based on

the Unified Modeling Language (UML)[8] standard notation for modeling software systems. MOF defines a standard set of concepts and notations for creating and using models, which can represent different aspects of a system. This includes concepts such as classes, attributes, operations, and relationships, which describe the system's structure and behavior.

Meta Models: A meta-model is a model of a model or a simplified version of an actual model of a system of interest or a software application that formally represents the structure and behavior of a particular model type in a formal and standardized way. Meta models are often used in Software Engineering (SE) to describe the structure and behavior of DSL or other modeling languages [37]. They can be used to understand how a model works or to improve the performance of a model by making it more efficient or accurate. Meta models are often used in Model Driven Development (MDD), focusing on creating and using high-level abstractions to design and implement software systems [38]. By using meta-models, developers can create a high-level view of a system and then use that view to generate code automatically, which can help improve the development process's efficiency and reliability. A model is usually defined as an abstraction of real-world entities, and a meta-model is another abstraction of the models. Similarly, metamodelling is analyzing and developing the rules, theories, and models helpful in constructing meta-models.

The **M0 layer** refers to the physical system or systems that are being modeled. It represents the *Real World* or the physical components, subsystems, and techniques that comprise the overall system being designed or analyzed. It is the foundation upon which the other layers of the MBSE models are built.

The **M1 layer**, built on top of the M0 layer, represents the functional requirements and system behavior. The functional requirements include system inputs, outputs, and interfaces, and the system's behavior includes its response to different inputs and conditions.

The **M2 layer**, built on top of the M1 layer, represents the system's architecture and design. This layer describes how the system's components, subsystems, hardware, and software interact. The M2 layer includes detailed specifications and design constraints such as interfaces, protocols, and data structures.

Finally, the **M3 layer** represents the system's implementation and verification. It includes the system's specific design details and implementation plans, including the detailed design of its components, selecting particular hardware and software, and testing and validation plans. This layer ensures that the system is built according to the specifications and design constraints defined in the M2 layer and meets the functional requirements and behavior described in the M1 layer.

[8] Unified Modeling Language https://www.uml.org/

After considering the information above, the meta-model can be further developed and refined. There are two approaches to designing a meta-model: *Top-down and Bottom-up* [36]. In the top-down method, we start with the overall process and then break it down into smaller steps. In the bottom-up approach, we begin with the individual steps and group them into more extensive procedures. *Top-down development* of a meta-model involves starting with a high-level representation of the structure and behavior of the models that the meta-model will describe (see figure 2.5). This high-level representation (M3 from figure 2.5) might include the overall design of the models (e.g., the types of elements and relationships that are allowed) and the rules and constraints that must be followed when creating and using the models. Once the high-level structure of the meta-model has been defined, it can be further refined and expanded upon by adding lower-level details (M2 and M1 and M0 levels). They include the attributes and operations allowed for each element, how elements can be related, and the semantics of the relationships between elements.

Bottom-up development of a meta-model involves starting with the specific elements and relationships that will be included in the meta-model and gradually building up to a higher level of abstraction. Once the lower-level details of the meta-model have been defined, they can be organized and grouped into higher-level concepts and structures, such as classes, packages, or packages of packages, to form the overall design of the meta-model [36].

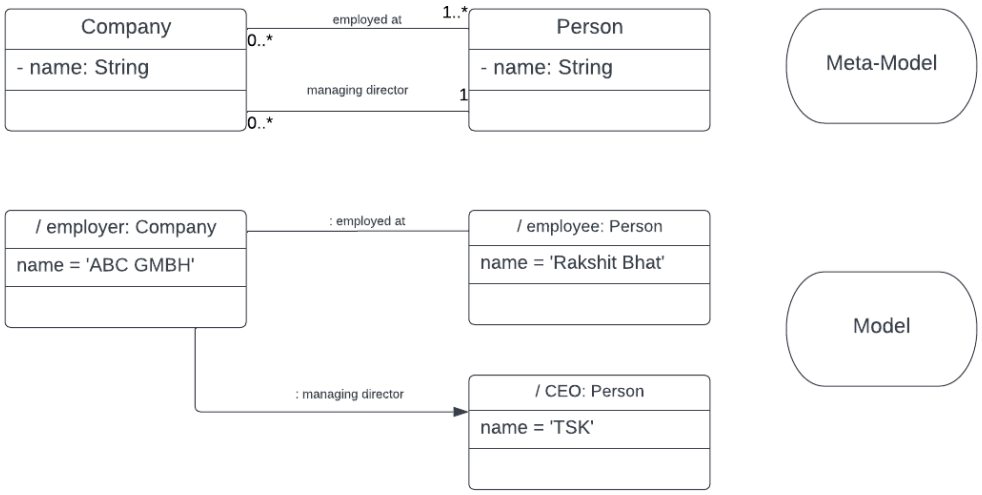


Fig. 2.6 Meta models and Models

Models: As shown in figure 2.5, a *Model* is a simplified representation of the real world or reality. Before any code is written, the model is a schematic that describes how the software

system should function. In software development, a model is a representation of a software system. It can be used to visualize the structure and behavior of a system or to analyze the system for potential issues or improvements. Models can take many forms, such as diagrams, graphs, or mathematical equations, and they can be created using various tools and techniques. Models are often used in MDD [38], focusing on creating and using high-level abstractions to design and implement software systems. In this approach, models are used as the primary source of design and implementation information, and the code is generated from the models using a code generator [39]. By using models, developers can better understand a system and identify potential problems or areas for improvement before they start writing code. From the example 2.6, we need to create models for every entity of reality (e.g., An employee and a CEO both are *Person* in meta-model but are separate entities in models). Using models, various adaptive model-driven user interface development systems are developed [40] and the authors defined twenty properties challenges for the Model-driven User interface and compared some tools that implement these properties. Similarly, modeling is a process and method of building models for some purpose or related to some domain. Therefore, models and modeling approaches are used to codify the UIs in different companies.

MBSE is a powerful approach to developing software that relies on using models to represent and analyze software systems. By using models to represent the various components and relationships within a software system, developers can more easily understand and analyze the system and make changes and updates more efficiently. However, these models do not emphasize offering visual notations to aid non-developers in creating such interfaces. Therefore, for our research, we use a recent method [39], which illustrates how to use low-code and Model-driven approaches to close the gap between designers and developers.

2.4 Task-based Usability Testing

The main focus of usability testing is that seeing someone use an interface is the best approach to determine what functions well and what doesn't. It would help if you offered the participants some assignments to observe them. The word "task" is commonly used to describe these assignments. Assigning tasks to the accurate number of participants can help determine the quality of the UI and the problems faced by the users. Overall, the UI design can be improved using the participants' feedback. Task-based usability testing is one way to determine the software's overall usability [41] by measuring the percentage of the tasks the users complete. These tasks need to be some scenarios, not just "*do something*", because it sets the users a stage for *why* they would perform the tasks. To get qualitative feedback from the participants, in [42], the authors provide *three good practices* and task-writing tips for designing better task scenarios.

***(1) Make the Task Realistic*:** So, the participants should be able to execute the tasks which could be completed efficiently and with the freedom to make their own choices. The participants will attempt to accomplish the assignment without genuinely interacting with the interface if you ask them to do something they wouldn't typically do. Therefore, it is necessary to create realistic tasks.
E.g., from *Videostreamer* application: The goal is to offer some movies that the user should watch.
Bad task: Watch a movie if the actor 'Mr. T' and actress 'Ms. K'.
Good Task: Watch a movie with more than 6.5/10 ratings.
In the example, the participants should be free to compare movies based on their criteria.

***(2) Make the Task Actionable*:** Here, the participants should be told what they need to do rather than how they would do it. These types of tasks help us determine if the task isn't actionable enough. E.g., if the participants tell the moderator, they cannot determine if they need to click on the link or if they are finding it hard to decide the next steps.
From *Videostreamer* application: The goal is to find a movie and show times.
Bad Task: You want to see a movie Sunday afternoon. Go to the app and tell where you'd click next.
Good Task: Use the app to find a movie you'd be interested in seeing on Sunday afternoon.
Therefore, if a participant removes their hand from the mouse and tells the facilitator they would first click in a specific place and then follow a link to where they want to go, it is a sign that the task is not straightforward or actionable enough.

***(3) Avoid Giving Clues and Describing the Steps*:** There are frequent hints about how to use the interface or the software in step explanations. These tasks must be more balanced with the users' behavior and give less valuable results. The participants should expose the navigation and some features on their own, giving accurate feedback about the interface. But, at the same time, we should try to include the words used in the UIs as they help the users navigate smoothly and would not lead to some confusion.

From *Videostreamer* application: The goal is to change the user's movie preferences.
Bad Task: You want to change your movie preferences. Go to the website, sign in, and change your movie preferences.
Good Task: Change your movie preferences from 'action' to 'comedy'.

Therefore, it is crucial to create a realistic test environment during usability testing [41]. It is also essential to provide the information required for the participant to complete the task without guiding them on specific actions. If the task scenario is unclear enough, the participant may ask for more information or confirm that they are on the correct path.

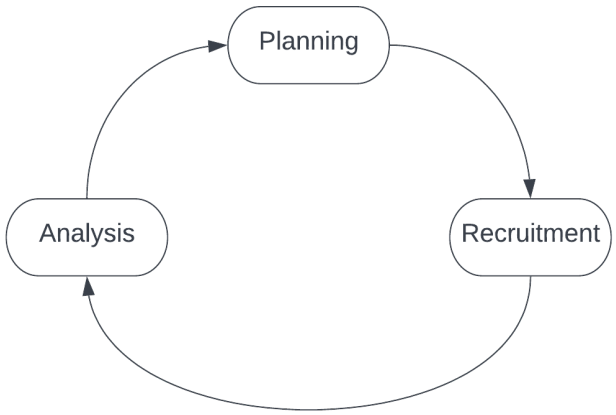


Fig. 2.7 Task-based Usability Testing steps

The fundamental principle of usability testing is to have actual users attempt to carry out essential tasks or assignments on your software, websites, mobile applications, or gadgets. For that, we divide the tasks into three steps.

Planning: In usability testing, planning is an important step. In this step, we decide on a process that fits our research question, hypothesis, and metrics. It is beneficial if we use mixed-methods services for a task-based usability study. The tasks and scenarios must be

well-defined in the planning phase. At the same time, we can create pre-study and post-study questions.

Recruitment: In this step, the users are assigned to the task as it is essential to select the correct number of user group sizes. There should be a proper way to interview the users before assigning them tasks so that the participants understand them correctly. The participants should also be able to ask questions or doubts while performing the honorarium tasks.

Analysis: Task-based studies analyze metrics, issues, and insights in great depth. For metrics, we calculate the task completion rates, task time, and task and test level perception questions. The problems that the participants encounter while performing the tasks need to be reported automatically to the development teams by sending screenshots, quotes, etc. There should also be a section to analyze some insights about the software that has worked well for the users while performing the tasks.

Overall, task-based usability testing is a tool for evaluating the usability and effectiveness of software systems. By focusing on specific tasks that users need to complete, we can gather valuable insights into how well the system supports these tasks and identify areas where we could improve the user experience.

2.5 Experimentation

In this section, we discuss the role that experimentation plays in the software development process and how designers can "prototype with real data" to improve the usability of the UI. Experimentation is a vital part of the design process for UI products. It involves testing different design solutions and variations to see which performs best in terms of usability, aesthetics, and other desired characteristics [7]. We can do this through various methods, such as usability testing, A/B testing, and rapid prototyping. Experimentation allows designers to iterate and improve their designs by creating different *'Software Variants'* and gathering data and feedback from real users. It is an important part of creating user-centered designs that effectively meet the needs and expectations of the target audience [6]. Experimentation helps product teams test out ideas early in the process with real-world consumers rather than settling on a single solution and executing it in the final phase [43].

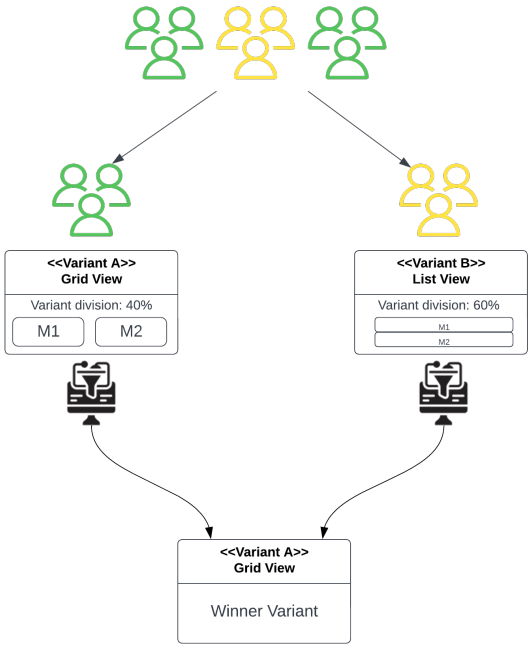


Fig. 2.8 A/B Testing

For conducting experimentation, various steps like *Users distribution*, *Continuous Experimentation (CE)*, *Variants distribution* exist.

User distribution: To conduct a successful experiment, the size of the study, or the number of participants, must be considered first. Statistics suggest that the more people you include in the investigation, the greater its statistical power, which impacts your level of confidence

in your findings [44]. Then, the participants should be allocated into groups at random. There are several levels of treatment given to each group (e.g., some prerequisites to the participants before experimenting). For assigning the subjects to groups, users are divided into a *between-subjects design* vs a *within-subjects design*. In a between-subjects design, individuals receive only one of the possible levels of an experimental treatment whereas, in a within-subjects design, every individual receives each of the experimental treatments consecutively, and their responses to each treatment are measured. In figure 2.8, a between-subject design is used where only one of the variants is assigned to the user.

Continuous Experimentation and Variants distribution: CE primarily aims to get users' feedback on the software product's evolution. As per figure 2.8, CE generally uses A/B testing in a primary case of comparing two variants, A and B, which are controlled and test variables in an experiment. First, the users or the participants of the study are separated into groups and are assigned one of the two variants. Since we have `GridView` and the `ListView`, one group of the users are assigned with List and one with Grid View (see figure 2.8). Then both users would be given some tasks (see Section 2.4), and the analysis of these tasks gives the winner among the variants. And in the end, developers make evidence-based decisions to direct the progress of their software by continuously measuring the results of multiple variants performed in an experimental context with actual users [45]. CE is an extension to the introduction of continuous integration and deployment, and all are summarized as constant software engineering [46]. Additionally, continuous experimentation can help designers and developers stay up-to-date with the latest design trends and best practices. By regularly testing and iterating on the design, they can learn from the feedback they receive and incorporate new ideas and techniques into their design. This can help keep the product or service relevant and competitive [7].

Overall, CE is a critical approach to take when developing UIs for products and services. It allows designers and developers to gather valuable feedback and data and use it to improve their products' design and User Experience (UX).

2.6 LEAN Development process

LEAN development is a software development methodology that emphasizes continuous improvement and eliminating waste to deliver customer value as efficiently as possible. It is one method within Agile development [47]. It is based on the principles of the LEAN manufacturing method, which was developed by *Toyota* in the 1950s and aims to eliminate waste and maximize value in manufacturing processes [48]. LEAN development's primary objective was to reduce loss, minimize waste, and encourage sustainable production. And therefore, as an Minimum Viable Product (MVP), in LEAN development, the product has the essential elements necessary to launch successfully and does not have to include any additional components.

In software development, we can apply LEAN principles to various aspects of the development process, including requirements gathering, design, coding, testing, and deployment [47]. The goal is to minimize waste and optimize the use of resources to deliver high-quality software products that meet the customer's needs.

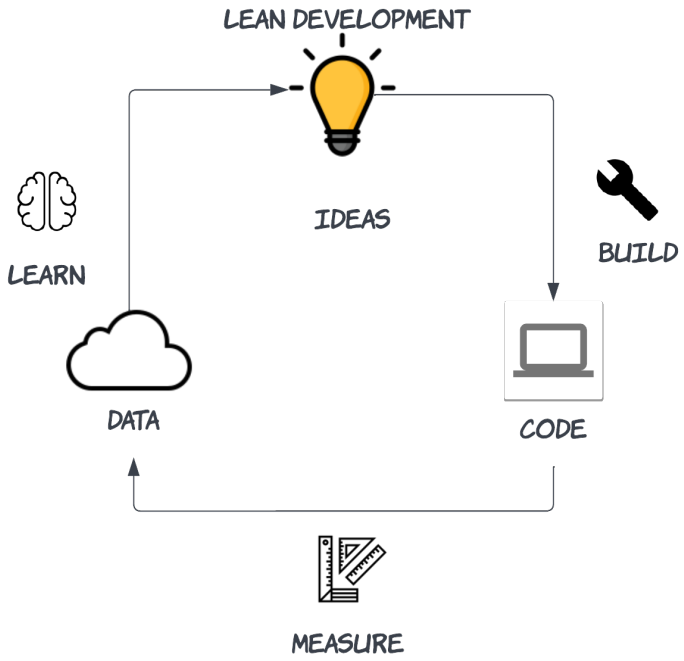


Fig. 2.9 LEAN development cycle[6]

[6]Image adapted from website: https://www.agile-academy.com/en/agile-dictionary/lean-development/

LEAN Development Cycle: The LEAN development cycle often includes three phases: *build, measure, and learn* (see figure 2.9). Steps like these are designed to help the development team continuously improve the product and development process.

Build: In the build phase of LEAN development, the development team works to build and deliver small increments of working software to the customer. This involves gathering requirements, designing, coding, testing, and deploying code [47]. The build phase's goal is to deliver customer requirements as quickly and efficiently as possible. To do this, the development team may follow model-driven development [49], continuous integration and delivery to ensure that the developed code meets the desired functionality and performance requirements.

Measure: In the measure phase of LEAN development, the development team collects data on the product's performance and the development process to identify areas for improvement [47]. It involves collecting data on customer usage and feedback, as well as data on the performance and efficiency of the development process. The measure phase aims to understand how the product is being used and its impact on the customer. It also identifies any bottlenecks or inefficiencies in the development process. The development team may use various tools and techniques, such as user testing, customer surveys, analytics data, and performance monitoring tools, to collect this data [50, 51]. They may also conduct regular reviews of the development process to identify improvement areas.

Learn: In the learn phase of LEAN development, the development team uses the data collected in the measure phase to identify areas for improvement and make changes to the development process and the product [47]. It may involve making changes to the product roadmap, adjusting development practices, or implementing new tools or techniques [52]. The goal of the learning phase is to improve the product continuously and the development process to deliver value to the customer more efficiently and effectively.

The final step in the cycle (see figure 2.9) would be to repeat the process, starting with identifying customer requirements and iterating through the *Build*, *Measure*, and *Learn*[9] phases again. Thus, the development team can constantly improve the product and the development process to deliver value to the customer more efficiently and effectively.

[9]LEAN cycle: https://www.einstein1.net/build-measure-learn/

As per [47], there are some key practices in LEAN development. These principles can be used to guide the discussion on which product development practices might be suitable for an organization, based on its specific circumstances.

Continuous delivery: Continuous delivery is the practice of regularly delivering small increments of working software to the customer rather than waiting until the end of the project to have a complete product. This allows the customer to see the project's progress and will enable them to give feedback and make changes as the project progresses. Continuous delivery helps to ensure that the software being developed meets the customer's needs and reduces the risk of delivering a product that does not meet their requirements [52]. It also allows the development team to identify and fix problems early in the process, saving time and resources. To practice continuous delivery, the development team must have a process for automatically building, testing, and deploying code changes [53].

Continuous improvement: Improving build quality in LEAN development refers to continuously improving the quality of the software being developed through effective coding practices, testing, and quality assurance processes [54]. To improve the build quality, we can use *Code reviews* [47], i.e., conducting regular code review sessions to help identify problems early on and ensure that code meets standards for readability, maintainability, and performance. Similarly, implementing *Quality Assurance* processes, such as testing, to catch and fix defects before they reach the customer can help improve the overall quality of the product.

Lean planning: It is a practice in LEAN development that focuses on maximizing value and minimizing waste in the software development process. It involves continuously prioritizing and reevaluating the work based on customer feedback to ensure that the team is first working on the most valuable features. As per [55], Lean planning aims to quickly identify and deliver the most valuable features to the customer while minimizing time and resources spent on non-critical tasks. This helps to ensure that the team is working on the right things and that the product meets the customer's needs.

In this way, LEAN development can help organizations deliver high-quality products to customers more efficiently and effectively.

In conclusion, the foundation chapter of the thesis is an integral part of the overall content. It provides context and background information on the topic being discussed and helps the reader understand the purpose and significance of the thesis. This section ensures the reader has the necessary information to understand and engage with the rest of the document completely.

Chapter 3

Design

The design of software systems is a critical aspect of SE, as it determines how the system will function and how well it will meet the needs of its users. The following chapter will detail the design and methodology used in the research and provide a thorough understanding of the research design and methods used in this study, and how it relates to the research question and objectives. This chapter will discuss the solution design (see section 3.1) which involves analyzing the problem by identifying the requirements and creating a detailed plan for how to address it. Then, in the next section, we explain our tool (see section 3.2) in terms of architecture, components, interfaces, and data models using the concrete requirements and solution approach developed in the previous section. This chapter will thoroughly explain the research design and methods used in this study and how it relates to the research question and objectives.

3.1 Solution Design

In software development, solution design is a step in the software development life cycle that defines a detailed technical design for building the software solution. This step is essential to ensure that the final solution meets the requirements and objectives and can be developed and tested efficiently. To ensure that our study results can be applied to other contexts, we have created abstract design knowledge based on the Design Requirements (DR) (see section 3.1.1), codify our knowledge into Design Principles (DP) (see section 3.1.2) and the overall solution design.

3.1.1 Design Requirements

Design requirements in Design Science Research (DSR) are typically defined as a set of constraints and specifications that must be met by the design artifact to be considered a successful solution. These requirements can be functional, such as the specific features or capabilities that the design must have, or non-functional, such as performance, usability, or scalability. To derive a solution, we define some DRs with the help of the literature review and comparison of some tools. In this context, each DR refers to a generalized requirement that can be standardized and applied to future software applications. These requirements are abstract and cover a wide range of software needs. We covered a wide range of topics including *UI Prototyping*, *Low-Code/No-Code development*, *Model-Based Software Engineering*, *Continuous Experimentation*, *Task-Based Usability Testing*, and the *LEAN development process*. The following section presents *eleven DRs* and their corresponding references in literature and tools.

DR1: Heterogeneous Users states that *The product should support diverse users with different needs, goals, and capabilities and integrate internal and external users.* In literature, this is reasoned by the fact that different users may have different needs, preferences, and levels of technical expertise [5]. By including a diverse group of users, you can get a broader range of feedback and insights into how the software performs for different users [23], simultaneously reducing the biases among developers [56]. In this context, the users can be from internal sources, such as employees, or external sources, like Amazon Mechanical Turk[1].

DR2: Iterative Refinement states that *The product should be designed and developed in an iterative, incremental approach with some continuous feedback mechanism.* In literature, this is reasoned by the fact that the iterative approach involves the idea of breaking down development into small, incremental cycles of work rather than trying to deliver a complete product all at once [47]. The key benefit of an iterative approach is that it allows the development team to get feedback from users and stakeholders early in the development process and to make adjustments [6] to the product as needed.

DR3: Interactive Design states that *User-friendly, frictionless, interactive and intuitive interfaces are essential for the software products helping the users improve the User Experience (UX).* In literature, this is reasoned by the fact that a tool should have easy navigation [28], clear and concise labels, icons, easy-to-use drag-and-drop functionality, collaboration

[1] Amazon Mechanical Turk: https://www.mturk.com/

capabilities (e.g., Figma[2] has a feature where more than one person can access the prototype), reviewing and refinement [57] features. Various live examples like Figma, Invision[3], Axure[4] have shown how user-friendly, interactive and intuitive tools have an impact on the users and improve the product.

DR4: Engage Stakeholders states that *Various stakeholders (e.g., designers, product managers, developers, etc.) must contribute to the quick development of the product.* In literature, this is reasoned by the fact that involving different project stakeholders and providing them with the necessary communication tools helps exchange and codify knowledge [23]. This makes sure that the stakeholders' perspectives, requirements [18], and their input helps shape the product's design.

DR5: Diverse Re-usable components states that *Collaboration between different team members and usage of the reusable components of various disciplines should make the tool easy for anyone to use on any platform and share their work quickly.* In literature, this is reasoned by developers' ability to create more consistent, user-friendly prototypes that adhere to the re-usable components' established style and principles [57] and help improve the overall design process and product development [28]. In many tools, we saw the use of many reusable components like buttons, textbox, and other UI components, data models for data transfer between components, and some guidelines to promote accessibility and usability of the tools.

DR6: Enhanced Evaluation states that *The software product should be evaluated using various means by having users perform specific tasks, collect feedback, and improve the product.* In literature, this is reasoned by the fact that testing of the GUI of a software application is done using functional and usability tests [42]. This helps the developers to identify any usability issues [58] and improve them continuously [17]. And this helps in the identification and preliminary validation of user requirements in the early stages of development [23].

DR7: Improved Accessibility states that *The tool should be a web-based application making it accessible and independent of any software, platforms (e.g., Macbooks, Windows Personal Computer (PC)s, Linux machines, Chromebooks) and have an auto-save feature to store the work on Cloud.* In literature, this is reasoned by the fact that Web-based tools have

[2]Figma Prototyping tool: https://www.figma.com/
[3]Invision: https://www.invisionapp.com/
[4]Axure Rapid Prototyping: https://www.axure.com/

several advantages [59] over traditional application-based tools. In many tools like `Figma`, `Invision`, and `Axure`, we saw features[5] like `Accessibility` i.e., can be accessed from any device with an internet connection, `Collaboration` i.e., have built-in collaboration features, making it easy for team members to work on the same prototype simultaneously, `Updates` i.e., automatically updated.

DR8: Combined feedback states that *The software product should gather user feedback and combine them to make improvements to the application.* In literature, this is reasoned by the fact that `Qualitative analysis` gathers an in-depth understanding of underlying reasons, opinions, and motivations [60]. Whereas `Quantitative analysis` measures and understands numerical data and helps identify patterns and trends [50]. And together, qualitative and quantitative analysis can provide a more complete picture of a situation and can be used to validate or disprove findings from one type of analysis [51].

DR9: Visualization states that *There should be a visualization tool for prototyping to help different stakeholders see and interact with the design and gather feedback.* In literature, this is reasoned by the fact that visualization helps in prototyping by allowing designers and stakeholders to see and understand the design in a way that is easy to understand [61]. It also allows designers to identify usability issues [17] early in the design process to make the end product user-friendly and easy to use. In tools, various methods, like creating `Wireframes`, `Mockups`, and `Interactive prototypes`, are used for visualization.

DR10 Quick delivery states that *A software product should be easily deliverable, reducing the complexity and deployment time and increasing the product's usability.* In literature, this is reasoned by the fact that when software is easy to build, development teams can be more productive and efficient, leading to lower development costs [30] and can be quickly delivered. It's also easy for different development team members to understand and work on the code, thereby improving collaboration [18] and generating high-quality code. Adding new features, scaling up, and making changes, as needed, make the tool more accessible and helpful [62] to ensure that the product is developed and deployed with minimum effort [62].

DR11 Classified UI versions states that *A software product should be tested with various UI versions or variants to analyze the best-fit variant for the product.* In literature, this is reasoned by the fact that a product can be tested against different design solutions and variations [46] to see which variant performs the best in terms of usability, aesthetics, and

[5] Some more features: https://redrocksoftware.com.au/10-benefits-of-web-based-applications-systems/

other characteristics [7]. At the same time, continuously improve [45] the product based on the feedback of the best-fit variant.

3.1.2 Design Principles

Design principles are guidelines or rules used to guide the design process in DSR [11]. They provide a framework for making design decisions [10] and help to ensure that the final solution meets the goals and objectives of the research. DPs are used for breaking down complex problems into smaller, more manageable parts. This section codifies our knowledge during the design study and derives DPs from abstract DRs from the iteration cycle of the DSR. The following shows the *Eight DPs* and references to literature and tools that build the foundation for the mapped DRs (see figure 3.1).

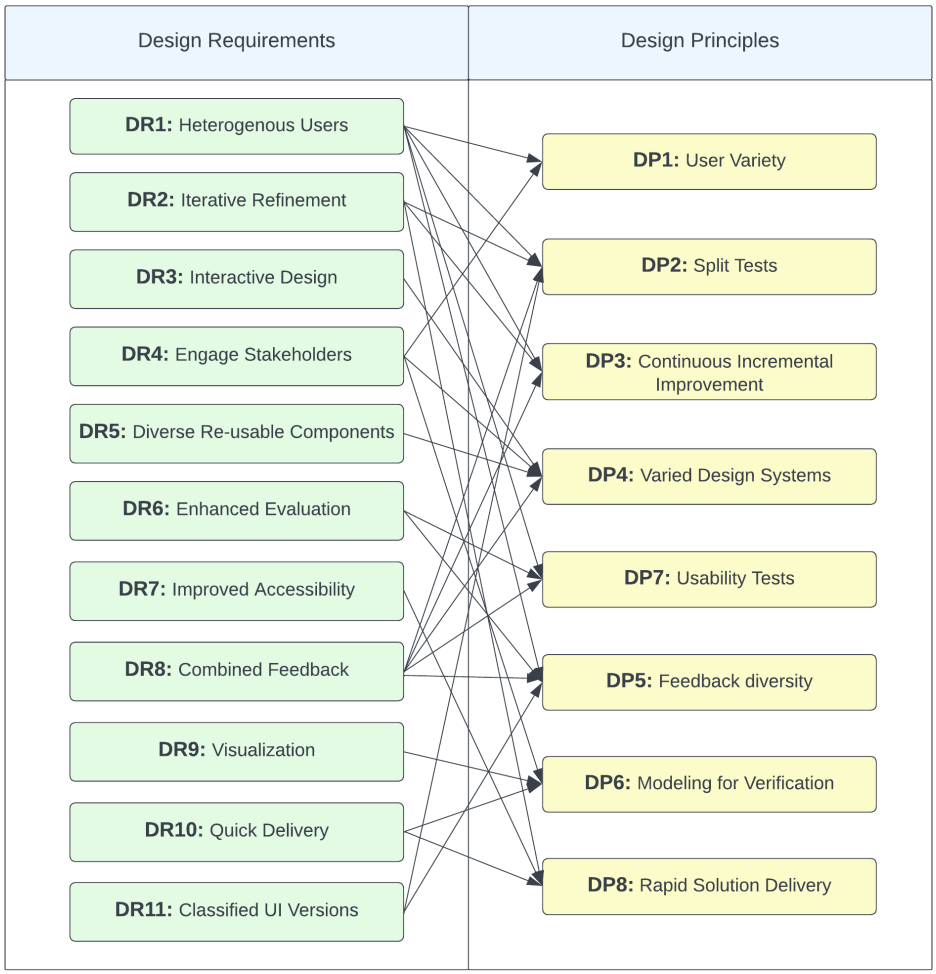


Fig. 3.1 A map between Design Requirements and Design Principles

DP1: User Variety: *Users with diverse requirements, objectives, and abilities should be able to use the product, and it should be able to accommodate both internal and external users within the organization.*

Developers have many unclear generalities early in the product development process [5] that they can clarify by testing the underlying assumptions using different types of users (i.e., *DR1: Heterogenous Users*), and involving various product stakeholders (i.e., *DR4: Engage Stakeholders*). This helps to gather the user requirements smoothly, thus improving the product's usability.

DP2: Split Tests: *Split tests can improve a software product by continuously testing different UI versions or variants with shuffled users to get a winner or a best-fit variant from the users' feedback and improve the product.*

Split Tests (also known as A/B Testing) are performed on a randomly divided sample group of different users (i.e., *DR1: Heterogenous Users*) by exposing each group to the UI of different versions (i.e., *DR11: Classified UI Versions*) to find out the feature that is most usable and functional for the users. The results of the test are then used to determine which version is more effective by combining the feedback (i.e., *DR8: Combined Feedback*) from the user groups and finally optimizing the whole product interatively (i.e., *DR2: Iterative Refinement*).

DP3: Continuous Incremental Improvement: *A software product broken down into minor, incremental phases in the software development process helps quickly improve software delivery and also helps to make product changes and improvements based on customer feedback.*

Using a continuous incremental approach, we can continuously improve software products by delivering value to customers as quickly as possible [48], constantly refining and improving the product [55], and delivering the product as soon as possible. Here, the iterative design should be used (i.e, *DR2: Iterative Refinement*) to get continuous feedback from a variety of users (i.e., *DR1: Heterogenous Users*). The feedback which is collected should be a combination (i.e., *DR8: Combined Feedback*) of various feedback helping significantly improve the application.

DP4: Varied Design Systems: *A product should be developed using Design Systems innovatively by the citizen developers without complete reliance and dependence on the IT department.*

A `Design System (DeSy)`[6] is a collection of reusable components governed by some standards that can be assembled in many ways to make different applications. A UI Prototyping tool is helpful when different interactive (i.e., *DR3: Interactive Design*) re-usable components are used as a drag-and-drop element (i.e., *DR5: Diverse Re-usable Components*) while creating the prototypes by engaging various stakeholders' contributions (i.e., *DR4: Engage Stakeholders*). It helps to get more feedback (i.e., *DR8: Combined Feedback*) from the users or the customers, improving the product's usability and Functionality.

DP5: Usability Tests: *Usability testing can provide valuable insights into how users interact with a product and can help to improve the UX. It's an effective method to detect usability issues early in the development process and to ensure the product is user-friendly, efficient, and easy to use.*

We can improve the usability of software by observing different users (i.e., *DR1: Heterogenous Users*) as they interact with the product and measuring how well they can accomplish specific tasks or goals (i.e., *DR6: Enhanced Evaluation*). Moreover, the users can provide valuable insights into how easy or difficult it is to use the product and help identify areas where developers could improve the UI or design using user feedback (i.e., *DR8: Combined Feedback*).

DP6: Feedback Diversity: *Diversity in software development feedback mechanisms helps ensure that a wide range of perspectives and experiences are considered when designing and testing software.*

Performing various tests (e.g., usability testing) with diverse users (i.e., *DR1: Heterogenous Users*) ensures that the software is accessible and easy to use for different groups of people with an accurate evaluation of the task provided to the users (i.e., *DR6: Enhanced Evaluation*). Moreover, diversity in software development feedback mechanisms (i.e., *DR8: Combined Feedback*), received by testing different UI versions (i.e., *DR11: Classified UI Versions*), helps ensure the software is inclusive and accessible.

DP7: Modeling for Verification *Modeling provides a clear and concise representation of the system being developed, thereby increasing communication among team members and stakeholders.*

A model-based approach increases transparency among various stakeholders, increasing their contribution to the product (i.e., *DR4: Engage Stakeholders*) due to their excellent

[6]Design Systems: https://www.invisionapp.com/inside-design/guide-to-design-systems/

visualization capability (i.e., *DR9: Visualization*). Furthermore, modeling tools can automatically generate code or other documentation from the models, which can help reduce errors, improve efficiency, and decrease development time (i.e., *DR10: Quick Delivery*).

DP8: Rapid Solution Delivery: *Rapid solution delivery in software development refers to the quick development and deployment of software solutions that meet the needs of users and stakeholders.*

We can achieve rapid solution delivery through the use of agile development methodologies and iterative design of the software (i.e., *DR2: Iterative Refinement*). With rapid solution delivery, software development teams can get feedback from users and stakeholders more quickly (i.e., *DR10: Quick Delivery*) and adjust from the feedback increasing accessibility (i.e., *DR7: Improved Accessibility*).

3.2 Solution Concept

3.2.1 Build

3.2.2 Measure

3.2.3 Learn

Chapter 4

Solution Implementation

4.1 Design Features

Chapter 5

Evaluation

5.1 User Case Study

5.2 Limitations and Risks

Chapter 6

Related Work

6.1 State of the Art Research

6.2 Comparison

3. Do not use ‘ditto’ signs or any other such convention to repeat a previous value. In many circumstances a blank will serve just as well. If it won’t, then repeat the value.

Table 6.1 A badly formatted table

	Species I		Species II	
Dental measurement	mean	SD	mean	SD
I1MD	6.23	0.91	5.2	0.7
I1LL	7.48	0.56	8.7	0.71
I2MD	3.99	0.63	4.22	0.54
I2LL	6.81	0.02	6.66	0.01
CMD	13.47	0.09	10.55	0.05
CBL	11.88	0.05	13.11	0.04

Table 6.2 Even better looking table using booktabs

Dental measurement	Species I		Species II	
	mean	SD	mean	SD
I1MD	6.23	0.91	5.2	0.7
I1LL	7.48	0.56	8.7	0.71
I2MD	3.99	0.63	4.22	0.54
I2LL	6.81	0.02	6.66	0.01
CMD	13.47	0.09	10.55	0.05
CBL	11.88	0.05	13.11	0.04

Chapter 7

Conclusion

7.1 Conclusion

7.2 Future Work

References

[1] William L. Kuechler and Vijay K. Vaishnavi. On theory development in design science research: anatomy of a research project. *European Journal of Information Systems*, 17:489–504, 2008.

[2] Faheem Ahmed, Luiz Fernando Capretz, and Piers Campbell. Evaluating the demand for soft skills in software development. *IT Professional*, 14(1):44–49, 2012.

[3] David J Teece. Business models, business strategy and innovation. *Long range planning*, 43(2-3):172–194, 2010.

[4] Alan M. Davis. Software prototyping. volume 40 of *Advances in Computers*, pages 39–63. Elsevier, 1995.

[5] Steve Blank. Why the lean start-up changes everything. https://hbr.org/2013/05/why-the-lean-start-up-changes-everything, May 2013.

[6] Eveliina Lindgren and Jürgen Münch. Raising the odds of success: the current state of experimentation in product development. *Information and Software Technology*, 77:80–91, September 2016.

[7] Aleksander Fabijan, Pavel Dmitriev, Helena Holmström Olsson, and Jan Bosch. The benefits of controlled experimentation at scale. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 18–26, 2017.

[8] Kathryn Whitenton. But you tested with only 5 users!: Responding to skepticism about findings from small studies. https://www.nngroup.com/articles/responding-skepticism-small-usability-tests/, 2019.

[9] Brian L. Smith, William T. Scherer, Trisha A. Hauser, and Byungkyu Brian Park. Data–driven methodology for signal timing plan development: A computational approach. *Computer-Aided Civil and Infrastructure Engineering*, 17, 2002.

[10] Shirley Gregor, David Jones, et al. The anatomy of a design theory. Association for Information Systems, 2007.

[11] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

[12] Jane Webster and Richard T Watson. Analyzing the past to prepare for the future: Writing a literature review. *MIS quarterly*, pages xiii–xxiii, 2002.

[13] Jasmin Jahić, Thomas Kuhn, Matthias Jung, and Norbert Wehn. Supervised testing of concurrent software in embedded systems. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 233–238, 2017.

[14] Faezeh Khorram, Jean-Marie Mottu, and Gerson Sunyé. Challenges and opportunities in low-code testing. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Jordi Cabot. Positioning of the low-code movement within the field of model-driven engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Mark A Hart. The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses eric ries. new york: Crown business, 2011. 320 pages. us$26.00. *Journal of Product Innovation Management*, 29(3):508–509, 2012.

[17] John D Gould and Clayton Lewis. Designing for usability: key principles and what designers think. *Communications of the ACM*, 28(3):300–311, 1985.

[18] Carlye A Lauff, Daniel Knight, Daria Kotys-Schwartz, and Mark E Rentschler. The role of prototypes in communication between stakeholders. *Design Studies*, 66:1–34, 2020.

[19] Jim Rudd, Ken Stern, and Scott Isensee. Low vs. high-fidelity prototyping debate. *interactions*, 3(1):76–85, 1996.

[20] Reinhard Sefelin, Manfred Tscheligi, and Verena Giller. Paper prototyping-what is it good for? a comparison of paper-and computer-based low-fidelity prototyping. In *CHI'03 extended abstracts on Human factors in computing systems*, pages 778–779, 2003.

[21] UXPin. What is a prototype. 2022.

[22] Mark van Harmelen. Exploratory user interface design using scenarios and prototypes. In *Proceedings of the Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*, pages 191–201, 1989.

[23] Paweł Weichbroth and Marcin Sikorski. User interface prototyping. techniques, methods and tools. *Studia Ekonomiczne*, (234):184–198, 2015.

[24] Figma high-fidelity prototype. https://www.figma.com/community/file/988854619809579450.

[25] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. *Human-computer interaction*. Addison-Wesley Longman Ltd., 1994.

[26] Ken Schwaber. *Agile project management with Scrum*. Microsoft press, 2004.

[27] Pedro Szekely. User interface prototyping: Tools and techniques. In *Workshop on Software Engineering and Human-Computer Interaction*, pages 76–92. Springer, 1994.

[28] G. F. Hoffnagle and W. E. Beregi. Automating the software development process. *IBM Systems Journal*, 24(2):102–120, 1985.

[29] Ender Sahinaslan, Onder Sahinaslan, and Mehmet Sabancıoglu. Low-code application platform in meeting increasing software demands quickly: Setxrm. In *AIP Conference Proceedings*, volume 2334, page 070007. AIP Publishing LLC, 2021.

[30] Alexander C Bock and Ulrich Frank. Low-code platform. *Business & Information Systems Engineering*, 63(6):733–740, 2021.

[31] Austin Miller. Low code vs no code explained. https://blogs.bmc.com/low-code-vs-no-code/?print-posts=pdf, 2021.

[32] Raquel Sanchis and Raúl Poler. Enterprise resilience assessment—a quantitative approach. *Sustainability*, 11(16):4327, 2019.

[33] Jordi Cabot. Positioning of the low-code movement within the field of model-driven engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Vagisha Arora. What are the three steps of low-code app development? https://www.planetcrust.com/what-are-the-three-steps-of-low-code-app-development, 2022.

[35] Christoforos Zolotas, Kyriakos C Chatzidimitriou, and Andreas L Symeonidis. Restsec: a low-code platform for generating secure by design enterprise services. *Enterprise Information Systems*, 12(8-9):1007–1033, 2018.

[36] Omg, meta-object facility. http://www.omg.org/mof/.

[37] Frédéric Jouault and Jean Bézivin. Km3: a dsl for metamodel specification. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

[38] Iyad Zikra. Implementing the unifying meta-model for enterprise modeling and model-driven development: An experience report. In *IFIP Working Conference on The Practice of Enterprise Modeling*, pages 172–187. Springer, 2012.

[39] Mariana Bexiga, Stoyan Garbatov, and João Costa Seco. Closing the gap between designers and developers in a low code ecosystem. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–10, 2020.

[40] Pierre A Akiki, Arosha K Bandara, and Yijun Yu. Adaptive model-driven user interface development systems. *ACM Computing Surveys (CSUR)*, 47(1):1–33, 2014.

[41] Frank Doesburg, Fokie Cnossen, Willem Dieperink, Wouter Bult, Anne Marie de Smet, Daan J Touw, and Maarten W Nijsten. Improved usability of a multi-infusion setup using a centralized control interface: A task-based usability test. *PloS one*, 12(8):e0183104, 2017.

[42] Marieke McCloskey. Turn user goals into task scenarios for usability testing. https://www.nngroup.com/articles/task-scenarios-usability-testing/, 2014.

[43] Miklos Philips. Evolving ux: experimental product design with a cxo. https://uxdesign.cc/evolving-ux-experimental-product-design-with-a-cxo-2c0865db80cc, 2020.

[44] Effie Lai-Chong Law, Virpi Roto, Marc Hassenzahl, Arnold POS Vermeeren, and Joke Kort. Understanding, scoping and defining user experience: a survey approach. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 719–728, 2009.

[45] Rasmus Ros and Per Runeson. Continuous experimentation and a/b testing: a mapping study. In *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 35–41. IEEE, 2018.

[46] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[47] Mary Poppendieck and Michael A Cusumano. Lean software development: A tutorial. *IEEE software*, 29(5):26–32, 2012.

[48] Freddy Ballé and Michael Ballé. Lean development. http://www.lean.enst.fr/wiki/pub/Lean/LesPublications/LeanDevBalleBalle.pdf, 2005.

[49] Nicolas Navet, Loïc Fejoz, Lionel Havet, and Altmeyer Sebastian. Lean model-driven development through model-interpretation: the cpal design flow. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[50] Forrest W Young. Quantitative analysis of qualitative data. *Psychometrika*, 46(4):357–388, 1981.

[51] Helena Holmström Olsson and Jan Bosch. Towards continuous customer validation: A conceptual model for combining qualitative customer feedback with quantitative customer observation. In *International Conference of Software Business*, pages 154–166. Springer, 2015.

[52] Eric Ries. *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Currency, 2011.

[53] Vidroha Debroy, Senecca Miller, and Lance Brimble. Building lean continuous integration and delivery pipelines by applying devops principles: a case study at varidesk. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–856, 2018.

[54] Sandra EP Silva, Robisom D Calado, Messias B Silva, and MA Nascimento. Lean startup applied in healthcare: A viable methodology for continuous improvement in the development of new products and services. *IFAC Proceedings Volumes*, 46(24):295–299, 2013.

[55] Tanja Suomalainen, Raija Kuusela, and Maarit Tihinen. Continuous planning: an important aspect of agile and lean development. *International Journal of Agile Systems and Management*, 8(2):132–162, 2015.

[56] Katrin Burmeister and Christian Schade. Are entrepreneurs' decisions more biased? an experimental investigation of the susceptibility to status quo bias. *Journal of business Venturing*, 22(3):340–362, 2007.

[57] L. Luqi and R. Steigerwald. Rapid software prototyping. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume ii, pages 470–479 vol.2, 1992.

[58] Kari Kuutti, Katja Battarbee, Simo Sade, T Mattelmaki, Turkka Keinonen, Topias Teirikko, and A-M Tornberg. Virtual prototypes in usability testing. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, pages 7–pp. IEEE, 2001.

[59] Michael Miller. *Cloud computing: Web-based applications that change the way you work and collaborate online*. Que publishing, 2008.

[60] Philipp Mayring. Qualitative content analysis: theoretical foundation, basic procedures and software solution. 2014.

[61] Urvashi Kokate, Kristen Shinohara, and Garreth W Tigwell. Exploring accessibility features and plug-ins for digital prototyping tools. 2022.

[62] Robert Waszkowski. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, 52(10):376–381, 2019. 13th IFAC Workshop on Intelligent Manufacturing Systems IMS 2019.

Acronyms

API Application Programming Interfaces

CE Continuous Experimentation

CSV Comma Separated Value

DB Database

DBMS Database Management System

DeSy Design System

DML Data Modeling Language

DP Design Principles

DR Design Requirements

DS Data Structure

DSL Domain Specific Language

DSR Design Science Research

GUI Graphical User Interface

HTML Hypertext Markup Language

IT Information Technology

LCDP Low-Code Development Platform

MBSE Model-Based Software Engineering

MDD Model Driven Development

MDSE Model Driven Software Engineering

MOF Model Object Facility

MQTT Message Queue Telemetry Transport

MVC Model-View-Controller

MVP Minimum Viable Product

NCDP No-Code Development Platform

OMG Object Management Group

PC Personal Computer

REST Representational State Transfer

SE Software Engineering

UAT User Acceptance testing

UCD User-Centered Design

UI User Interface

UML Unified Modeling Language

UX User Experience

Appendix A

How to install our Low-code Application

Windows OS

TeXLive package - full version

1. Download the TeXLive ISO (2.2GB) from
   https://www.tug.org/texlive/
2. Download WinCDEmu (if you don't have a virtual drive) from
   http://wincdemu.sysprogs.org/download/
3. To install Windows CD Emulator follow the instructions at
   http://wincdemu.sysprogs.org/tutorials/install/
4. Right-click the iso and mount it using the WinCDEmu as shown in
   http://wincdemu.sysprogs.org/tutorials/mount/
5. Open your virtual drive and run setup.pl

or

Basic MikTeX - TEX distribution

1. Download Basic-MiKTEX(32bit or 64bit) from
   http://miktex.org/download
2. Run the installer
3. To add a new package go to Start » All Programs » MikTex » Maintenance (Admin)
   and choose Package Manager

4. Select or search for packages to install

TexStudio - TEX editor

1. Download TexStudio from
   http://texstudio.sourceforge.net/#downloads

2. Run the installer

Mac OS X

MacTeX - TEX distribution

1. Download the file from
   https://www.tug.org/mactex/

2. Extract and double click to run the installer. It does the entire configuration, sit back and relax.

TexStudio - TEX editor

1. Download TexStudio from
   http://texstudio.sourceforge.net/#downloads

2. Extract and Start

Unix/Linux

TeXLive - TEX distribution

Getting the distribution:

1. TexLive can be downloaded from
   http://www.tug.org/texlive/acquire-netinstall.html.

2. TexLive is provided by most operating system you can use (rpm,apt-get or yum) to get TexLive distributions

Installation

1. Mount the ISO file in the mnt directory

   ```
   mount -t iso9660 -o ro,loop,noauto /your/texlive####.iso /mnt
   ```

2. Install wget on your OS (use rpm, apt-get or yum install)

3. Run the installer script install-tl.

   ```
   cd /your/download/directory
   ./install-tl
   ```

4. Enter command 'i' for installation

5. Post-Installation configuration:
   http://www.tug.org/texlive/doc/texlive-en/texlive-en.html#x1-320003.4.1

6. Set the path for the directory of TexLive binaries in your .bashrc file

For 32bit OS

For Bourne-compatible shells such as bash, and using Intel x86 GNU/Linux and a default directory setup as an example, the file to edit might be

```
edit $~/.bashrc file and add following lines
PATH=/usr/local/texlive/2011/bin/i386-linux:$PATH;
export PATH
MANPATH=/usr/local/texlive/2011/texmf/doc/man:$MANPATH;
export MANPATH
INFOPATH=/usr/local/texlive/2011/texmf/doc/info:$INFOPATH;
export INFOPATH
```

For 64bit OS

```
edit $~/.bashrc file and add following lines
PATH=/usr/local/texlive/2011/bin/x86_64-linux:$PATH;
export PATH
MANPATH=/usr/local/texlive/2011/texmf/doc/man:$MANPATH;
export MANPATH
```

```
INFOPATH=/usr/local/texlive/2011/texmf/doc/info:$INFOPATH;
export INFOPATH
```

Fedora/RedHat/CentOS:

```
sudo yum install texlive
sudo yum install psutils
```

SUSE:

```
sudo zypper install texlive
```

Debian/Ubuntu:

```
sudo apt-get install texlive texlive-latex-extra
sudo apt-get install psutils
```

Appendix B

Definitions and explanations of terms

Wireframes: A wireframe is an essential visual representation of the layout and structure of a product, showing the placement of text, images, and buttons.