



**POLITECNICO**  
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE  
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

## Dal design al codice: generazione automatica della componente front-end di un'applicazione mobile

LAUREA MAGISTRALE IN COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

**Author:** MONICA BUTTIRONI

**Advisor:** PROF. LUCIANO BARESI

**Academic year:** 2021-2022

### 1. Introduction

In a dynamic and competitive market such as that of mobile apps, it is essential to guarantee both the quality of the product and the speed of its release on the market [1]. In this context, the multiplicity of tools required by the implementation process represents a criticality, as it increases the quantity of required skills. A first solution path, undertaken by all big tech companies, consists in adopting cross-platform technologies for the implementation process. It is, however, possible to further increase information sharing by designing tools that directly tie design to implementation. The objective of this thesis work is to demonstrate the possibility of realizing such a tool with modern technologies, and to evaluate its performance from the point of view of both development time compression and quality of the final result, i.e. of the product code.

### 2. Solution overview

One of the possible ways to solve the presented problem is to build a modular and configurable pipeline capable of reading the UI/UX design document and generating, in whole or in part, the corresponding source code. Modularity is a key element for the applicability of the tool and

the approach both to the varied current context and to future developments, while configuration flexibility allows the pipeline to adapt to even very different projects. The founding element of this approach is the possibility of defining an abstract formal model of the application, independent of the design tool or the technology chosen for implementation. The model-based pipeline proposed in this thesis work is inspired by the LLVM compiler framework [4] and is composed of three functional units: a first module (**front-end**) converts the design document into an instance of the abstract model, which assumes the role of intermediate representation, a second module (**middle-end**) manipulates this model without changing its formal structure, and a third module (**back-end**) converts it into source code.

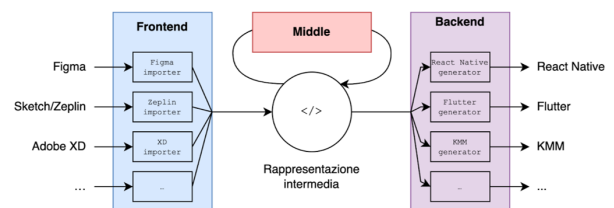


Figure 1: Solution architecture

### 3. Technology selection

Currently, there exist many multimedia graphics editors, which differ in availability, compatibility with the most popular platforms, and offered features. Some of them, like Adobe Illustrator, are general-purpose graphical editors, while others, such as Figma or Sketch, are built specifically for web and mobile UI design and offer dedicated features like interactive prototypes and component-based modeling.

On the implementation technologies side, the landscape is split between native development and cross-platform solutions. Native tools allow for maximum runtime performance, while cross-platform solutions are aimed to improve and optimize the development process, shrinking the required time and skills.

In order to prove the feasibility of the described approach, this thesis work will focus on the conversion from Figma design documents to React Native code. Figma has been selected because it is the most used design tool for mobile apps and it exposes a well-documented public REST API [2]. React Native [3], on its side, has been chosen because it has been the most used cross-platform framework for years and because of its extremely rich ecosystem.

### 4. Formal model of mobile apps

The adoption of a model-based approach for generating mobile applications requires defining the model in a formal and structured way. This definition plays a central role because the model constitutes the interface between the three modules that make up the proposed solution and is the starting point for their implementation. To allow for maximum interoperability, this model has been designed to be natively serializable to JSON.

The main concepts of the model are:

- Nodes
- Styles
- Components
- Screens

#### 4.1. Nodes

The user interface of a mobile application can be seen as a collection of elements organized in a tree-like structure, where they represent the nodes. Through the observation of existing mo-

bile applications, it is possible to identify five node classes:

- **Frame**: usually used as a container for other elements, to which it provides a background surface and a reference for positioning
- **Vector**: scalable vector graphic primitive
- **Text**: text element
- **Instance**: an instance of a defined and reusable component
- **Group**: logical group of nodes, without own visual functionality

All node classes share a common structure, which contains general-purpose properties, but they can also have some type-specific attributes.

#### 4.2. Styles

The second key concept of the intermediate representation is the definition of styles. These can be classified into three categories: **color** includes styles that affect color (solid colors, gradients, and images), **text** includes styles that define the typographic aspect of the text (font family and size, etc...), while **effect** contains styles that define accent elements (such as shadows or blurs). There exists also the possibility of defining a global table of styles that can be shared between multiple nodes and referenced via a unique alphanumeric identifier. Each node (either part of a screen or a component) can define its unique style or reference a global one.

#### 4.3. Components

The concept of components, common to both the main UI design tools and cross-platform technologies, allows you to design modular, extensible, and reusable interfaces. They, therefore, represent a top-level concept of the model. Each component is defined by a tree of nodes, which represents its structure, and by a domain of variability, which allows parameterizing some aspects. The domain of variability allows you to see the components as visual units with their own semantics, and to vary some minor graphic aspects. Each instance of the component is therefore a variant of it: the differences with respect to the main definition are recorded as points of variability, while the structure remains fixed (ie the type of nodes used and their nesting).

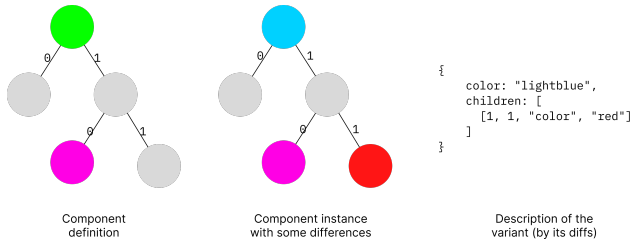


Figure 2: Component and instance definition

#### 4.4. Screens

Screens are the navigation units of a mobile application and they are usually rendered as fullscreen containers. Each screen is built assembling nodes in a recursive way, hence they can be modeled as node trees just like components (but without variants, as screens are stand-alone top-level units).

### 5. Front-end module

Each of the components of the front-end module of the proposed solution has the task of obtaining the data of the abstract model of an application from a specific UI/UX design software and processing them to make this model conform to the intermediate representation. The implemented module, related to Figma, performs the following operations:

- **Project download:** download design data from the public API exposed by Figma, which provides a structured representation of all elements used in the UI design document, a table of defined styles, and a table of components
- **Images download:** retrieve image URLs from the public API exposed by Figma
- **Pages selection:** remove any content that should not be involved in the code generation
- **Node properties elaboration:** analysis of data obtained from Figma and conversion of node properties from Figma format to the intermediate representation
- **Styles elaboration:** analysis of the style data of each node and construction of the global table of shared styles. For each style, the definition and its use within the project nodes are identified
- **Components elaboration:** analysis of the structure of the components defined in Figma, construction of the table of components and search and indexing of all their

variants. Since a component can contain, in its definition, an instance of another one, it is necessary that the analysis follows an order compatible with the dependency relationship of the components.

### 6. Middle-end module

The middle-end module performs a series of transformations and optimizations on data structured according to the intermediate representation, keeping its structure constant. In particular, this module is organized into independent units, which do not require a particular order of execution in order to operate correctly. In fact, they are able to act both starting from data coming directly from the front-end module and starting from the processing result of any other unit. Since the definition of the middle-end units depends exclusively on the structure of the intermediate representation, they can be used in any design-to-source-code conversion pipeline, regardless of the technologies involved in the other two modules. This choice favors the scalability of the tool, making possible a future extension with new features and optimizations.

As a proof of concept, the following units have been developed:

- **Sum consequent solid colors:** using the color-blending techniques, solid color layers in the same style definition are blended into a unique one
- **Style definitions decoupling:** merge duplicate shared styles definitions
- **Find non-unique inline styles:** search for inline styles that are shared among different nodes and convert them to shared global styles
- **Sum consequent groups:** nested groups with the same properties get squashed

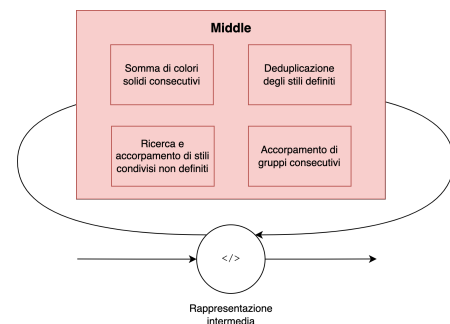


Figure 3: Middle-end structure

## 7. Back-end module

The back-end module of the solution proposed in this thesis has the task of converting the abstract model of an application, encoded in terms of the intermediate representation, into executable code. The ultimate goal is to generate a code that is as complete as possible with respect to the information available at the prototype/design document level. The React Native project generated by the backend module developed for this thesis work is based on the Expo framework, which gives a solid foundation and completes the core React Native API.

The steps taken by the code generator are the following:

- **Project scaffolding:** generation of project files and creation of folder structure
- **Shared styles generation:** generation of the code corresponding to the table of globally defined styles, through the use of style sheets, where possible, or dedicated components
- **Components generation:** generates the source code of the various components, as defined in the components table. Each of them exposes suitable properties to allow the implementation of all its variability points. The visual part is generated by recursively iterating the component tree, inserting the defined properties.
- **Screens generation:** takes care of transforming the definition of the screens into executable code. For each screen, a dedicated folder is generated in order to contain its definition and the those of all the non-shared styles used. Just like components, code generation occurs by recursively iterating the tree structure of each of them

Each generated file is complete of all the import statements, syntactically correct and formatted according to the recommended best practices, in order to allow for best readability and customization by a developer.

## 8. Evaluation method

In order to assess the quality and effectiveness of the described approach, it is possible to run it against an example test case. This test case should be both complete, with respect to the capabilities of the tool, and yet simple, to al-

low manual inspection and analysis of the result. Moreover, it should be a design document produced by an independent third-party, to avoid biasing the results and to demonstrate the universality of the approach.

The interesting metrics to be evaluated are:

- **Performance:** it is possible to compare the time required by the generator to convert the design document to code with the time required for a developer to carry out the same task
- **Generated code quality:** it is possible to run syntax checks on the generated code as well as to evaluate its readability and ease of modification
- **Semantic content of the generated code:** it is possible to estimate how much of the semantics expressed by the design document has been captured by the tool and converted in runnable code
- **Completeness of the generated code:** it is possible to check how much of the information contained in the design document has been used in code generation

## 9. Conclusions

The described approach appears to fulfill the initial goal, as it certifies the possibility of creating a completely automated tool to convert a design document in executable code. It must be taken into account that the quality of the final result heavily depends on the quality of the design document: a rich and well-organized style guide and a component-based approach perform much better than a copy-pasted bunch of canvases. It can also be stated that generating the code automatically is more efficient than having a developer write it. Automatic code generation may also help learning new technologies, as it produces a coherent and structured codebase, based on well-known patterns for the specific technology. Code generation is not, however, a complete solution for app development: a professional developer analysing a UI design document is able to make semantical assertions, while an automated generator is limited to the understanding of shapes, geometry and colors.

Future extensions of the work may integrate other design tools or backend language, or explore the possibility of generating some code related to navigation primitives and patterns.

## References

- [1] Digital 2022 – i dati globali, 2022. URL: <https://wearesocial.com/it/blog/2022/01/digital-2022-i-dati-globali/>.
- [2] Figma api, 2022. URL: <https://www.figma.com/developers/api>.
- [3] React native, 2022. URL: <https://reactnative.dev/>.
- [4] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.