Applying EE Theories to Component-Based Software Design and Development

Ondřej Dvořák
Department of Software Engineering,
Faculty of Information Technology,
Czech Technical University,
Prague, Czech Republic,
ondrej.dvorak@fit.cvut.cz

Abstract. Component-Based Software Development is a promise that was made in 1968 during the NATO symposium, where “software engineering” term is said to have originated. Since then, the ideal state of software development as an engineering discipline is “assembling instead of programming”. However, even today, we seem to be quite far from this ideal. The Enterprise Engineering theories deal with function and construction of an enterprise in an implementation-independent way. We argue that the theories may be successfully applied for the software engineering domain to improve component-based design.

Key words: Enterprise Engineering, Software Engineering, DEMO

1 Introduction

1.1 Motivation

The goal of my doctoral thesis is to propose an improvement of component-based applications design. I started my research across normalized systems and Enterprise Engineering (EE) theories. In my first year of Ph.D. studies, I found an interest in lectures about enterprise engineering. I was introduced to DEMO (Design & Engineering Methodology for Organizations) that is formally grounded in EE theories. The theories were developed and practised by CIAO! Network (www.ciaonetwork.org). Some of them gave me an impression they could be easily mapped to software development. Thus, even a component-based design of applications might possibly benefit from them. To prove this idea I decided to analyse most of the theories and consider their impact on an overall software engineering. At first, I mapped the PSI theory into so-called confirmation engine, and I implemented it in a commercial component-based application platform CoRiMa. A corresponding paper was already accepted at CBI 2015 [6]. The evaluation of an influence of all EE theories is not yet finished, nevertheless, the ideas behind are introduced in this paper.
1.2 Research Overview

"Assembling instead of programming" is a paradigm that Douglas McIlroy brought at the 1968 NATO Conference on Software Engineering [17]. Researchers and developers have tried for years to turn that dream into reality by organising a software into modules.

An overview of these attempts can be found in [10]. First, languages supporting structured programming were designed. The modularity was achieved by aggregating both data variables and instructions. Second, object-oriented programming, where classes built new level of modularity, was introduced. Third, libraries and packages emerged. Their purpose was to collect compiled reusable code constructs. Finally, component-based development was proposed. It perceived a component as a modular layer on top of objects in object-oriented languages [10]. Szyperski [22] defined a component as “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties”.

However, after decades of modular approaches in software development, we seem to be quite far from the McIlroy’s ideal. The modular architecture must bear other challenges, e.g., high coupling, and low cohesion. These characteristics of software might influence its evolvability. Thus, they might influence overall costs for a maintenance of the software. Although few metrics for measuring and rating quality of software were proposed [23], a research focused on so-called Normalized Systems (NS) theory [11] seems to be a cornerstone for tackling evolvability of a software. To exhibit high evolvability that theory requires information systems to be free from so-called combinatorial effects [11].

Next to the information systems (IS), there exists an interesting group of systems called enterprises. Giachetti [7] defines them as “... complex, socio-technical systems that comprise interdependent resources of people, information, and technology that must interact with each other and their environment in support of a common mission”. A range of EE theories were presented to support an intellectual manageability of enterprises. Both enterprises and IS are systems. It seems there might be an overlap between them. Thus, we believe that the same EE theories might contribute a software development.

1.3 Goals and Methodology

The goal of this paper is to introduce a research focused on applying EE theories to software engineering (SE).

In section 2, we give an overview of similarities between enterprises and software systems. We categorise the foundational EE theories into philosophical, ontological, technological and ideological, and we roughly explain each theory
that seems to have an overlap to software engineering. We elaborate how it may contribute to SE, and at the end, we show how they fit into a future component-based framework.

In section 3, we clarify a current state of the art of component-based software. We outline how EE theories might contribute to this field. In section 4, we describe a relation to normalized systems. We show the differences, and we outline how normalized systems may influence our research. We conclude our paper in section 5, where we outline next steps in our research.

2 Overlap of EE theories and SE

As already cited Giachetti’s research, enterprises can be seen as big systems comprising interdependent resources of people, information, and technology. While taking the role of a human in such a system into account, and while considering a uniqueness of people from their communication point of view, each enterprise must be unique and substantially complex, too. Because of our limited cognitive faculties, the complexity of an enterprise usually exceeds our ability to deal with that directly. Instead, we need to handle the complexity by abstracting and modelling. A range of methods and notations like BPMN [20], FlowChart [21] or Archimate [22] were developed for that purpose. Unfortunately, not all of them do really abstract from a complexity of an enterprise effectively. On the other hand, DEMO (Design and Engineering Methodology for Organizations [5]) method puts emphasis on an intellectual manageability of enterprises while concentrating on their essence - the social interactions. DEMO has a strong theoretical background represented by enterprise engineering (EE) theories like τ-theory (TAO) [16] or ψ-theory (PSI) [15]. These are the objective of this paper.

Our hypothesis is that big software systems can be seen as a sort of enterprises. Thus, the same EE theories and methodologies can contribute to their understanding and might be applicable to their engineering. Beside others, the complexity of an enterprise is given by a diversity of options people can use to interact and by the expressive abilities of a language with which they communicate. Similarly, the complexity of a software system is influenced by the offer of various languages, libraries, and frameworks. This gives a considerable architectural freedom to all developers. They can use their skills and express thoughts in a form of a source code. As a business-process architect can reduce the complexity of an enterprise by introducing guidelines for a human interactions, a software architect can set up guidelines for a software implementation to reduce inconsistencies in a source code. A common mission of an enterprise is to successfully deliver and maintain its products while reducing costs on resources it operates with. Equally, a common mission of a software system is to provide features expected by their users while reducing costs on the code typing and on its maintenance.
In this section, we would like to elaborate on these similarities between enterprises and software systems. We would like to propose a way by which the EE theories could be applied on a software systems development.

2.1 EE Theories

The EE theories scrutinised in this paper were introduced in [Dietz, Hoogervorst et. al. 2013]. Its most current version is on the [Figure 1]. The theories are categorised into philosophical, ontological, technological, and ideological. These at the bottom influence the theories on the top. For example, the philosophical theories represent foundation of all the others. In this paper, we do not concentrate on an influence of ideological theories to a software engineering. Indeed, e.g., political philosophy of J. Habermas laid down the origins of enterprise engineering. However, we do not dare to elaborate it in the current stage of our research.

We start with the foundations of philosophical theories. We continue with ontological, and we end with technological theories. Each is introduced together with an explanation of its influence on SE.

![Figure 1: Classification scheme with EE theories](image)

2.2 Philosophical Theories

Several philosophical theories like mathematics and logic have obviously an influence on a software development. Nevertheless, their specific influence on SE
is out of scope of this paper. We do not elaborate it any further. We focus on two philosophical EE theories, $\phi$-theory and $\tau$-theory that are categorised as philosophical theories. We introduce them and consider how SE can benefit from them.

**The $\phi$-theory** (FI) (Fact and Information) is a theory about a knowledge in general. It sees the fact as a part of an elementary thought, which in addition consists of an intention. As Dietz clarifies, “every concrete system (e.g., organisation) has an associated world ... A world consists of things and someone’s knowledge about these things consists of facts.” Information is an intermediary for exchanging thoughts between humans.

The $\phi$-theory is rooted in a *semiotic triangle* presented on the Figure 2 (left). It was introduced by Ogden and Richards. Dietz presents its slightly adopted version in [14]. In our minds, there are *thoughts*. Thoughts refer to concrete things called *referents*, and they are expressed in *signs*.

Nevertheless, the semiotic triangle is quite a simplified representation of the three core concepts in $\phi$-theory: thought, sign and referent. A more sophisticated framework for studying thoughts (information) was proposed by Ronald Stamper. It is called *semiotic ladder*, and it is shown on the Figure 2 (right). Its main focus is on the thought and the sign. On the top of that ladder, there is a *social world* which is dealt in a $\psi$-theory (section 2.3). It investigates how intentions are related to commitments of social individuals. The semiotic triangle studies a content of thoughts and is divided into semantics and pragmatics. *Semantics* inspects a meaning of a sign (or a sentence) in some language. *Pragmatics* is about an intention of sharing the thought among subjects. A form of a thought is also divided into two parts: syntactics and empirics. *Syntactics* deals with a form of a sentence that must respect formalisms and rules given by a grammar. *Empirics* studies how to express parts of sentences (e.g., words can be written in Roman letters or in Morse code) in a form of codes and patterns. Finally, a *physical world* does not belong to the field of semiotics. It contains traces and substances inscribing patterns and codes above.

Although the $\phi$-theory belongs to EE theories, we believe it can be easily seen as a software engineering theory. The software engineering is about transforming objects (*concrete objects*) from a concrete world to conceptual objects. These are represented by symbols understandable for software. Software operates in a world of symbols and semantics. For example, a given programming language has its world of language elements (variables, expressions, exceptions, flow constructs, etc.). Their composition into grammatically correct sentences has a specific meaning for a computer.

The challenge of software engineering is to map the concepts introduced by $\phi$-theory into symbols and concepts of a computer. The final behaviour should match. On the Figure 3 (left), a semiotic triangle is shown from an ontological point of view. The referents are considered as concrete objects. They are repre-
presented in our minds as conceptual objects. Names are signs by which humans signify the conceptual objects. Conceptual objects in a human world are mapped into conceptual objects in a context of a computer. Names in a human world are mapped into symbols of a computer. It is built using a range of paradigms, programming languages, libraries and various technologies supporting software development.

Fig. 3: Mapping of a semiotic triangle

Needless to clarify that the mapping is not all the time one-to-one relationship. Our current research is focused on how to do this mapping more deterministically and how to take all the aspects of the semiotic ladder into account.

The τ-theory (TAO) (Teleology Across Ontology) is a theory about a relation between subjects with purposes and objects with properties. Such a subject-object relationship is called an affordance. Dietz explains an ontological point of view in τ-theory as: “... objects are studied as they are completely disregarding the purpose(s) subjects could use them for...” [16]. Conversely he reveals the tele-
logical point of view where objects are studied as purposive entities. Subjects (as human beings) “…observe objects, create and use objects…” [16] to satisfy their needs and desires. From the Theory of Affordances [8], these needs and desires are not satisfied by using objects and their properties, but rather by affordances that the objects offer.

Affordance is a bridge between the ontological and teleological point of view. Its core notion is explained by Dietz in [16] on an example below. We illustrate it on the Figure 4.

“If you (subject) want to sit (purpose), you may perceive that you can sit (affordance) on a tree-stump (object), because the height of its surface (property) fits your purpose. So, whereas the purposes of subjects are purely subjective, and the properties of objects are purely objective, an affordance is a subject-object relationship. Because of the unlimited imagination of the human mind, the number of affordances of an object is basically unlimited. Note that the being subjective of an affordance implies the abilities of the subject: for a 2-year old child, the above mentioned tree-stump is not ‘sit-on-able’, and for a physically disabled person, a ladder is not ‘climb-able’”

Fig. 4: Core objects of study in τ-theory [16]

Next important notion of τ-theory is a notion of functions. The needs and desires of a subject are satisfied by using natural objects. Beside that, the subject may create artefacts. Again Dietz explains in [16]:

“Artefacts are designed and made with some affordance in mind, which is commonly called the function of the artefact. Examples: the function of a chair is to sit on (a chair may offer the affordance sit-on-able), and the function of a table is to sit at (a table may offer the affordance sit-at-able).”

Last but not least notion of τ-theory is a notion of constructions. The given function of an artefact is viable thanks to its construction. A construction refers
to a way of how the artefacts are created, which parts and substances are used, and how the specific parts are interconnected. For example, a constructional perspective on a car illustrated in [16] is characterising a car as a set of inherent properties (e.g., a dimension or a mass). It can be disassembled into pieces. The important note is that there is only one way to do it. We end up with the same set of parts after disassembling the car.

Contrary, a function decomposition of a car is purely subjective. Two different people can identify different affordances offered by the car. It depends on the purpose they want to use it for. Thus, there can be more functional decompositions of the same car. See the construction vs function perspective on a picture Figure 5.

Fig. 5: Construction (left) vs Function (right) perspective [16]

In the previous section, we discussed φ-theory. From the SE point of view, its focus is on a mapping of a reality into concepts of a computer Figure 3. Similarly, τ-theory is a relation between a subject and a software. Nevertheless, the subject does not necessarily need to be a human. It can be software component in a relation to another software component (library, service, etc.) as well. Thus, τ-theory concentrates on both, user-software and software-software relationships. We can identify a range of aspects in τ-theory that might affect software design, especially component-based design. By the component we mean a software artefact (library, service or even a class). Such a component provides an interface through which other components can utilise its capabilities. A component with properties can satisfy needs of a user (or another component). τ-theory sees artefacts as objects designed and created by humans. From SE point of view, these artefacts are mostly created by programmers. All in all, SE might define a relation between a subject and its purpose and a software component with its properties. Since this definition is analogical to a definition of an affordance in τ-theory, we think the same notion can be applied in SE. Thus, we argue, an interface can be considered as a representation of an affordance.

We understand a function defined in τ-theory as a way of using a given affordance. From this perspective, the function in SE could be a method of an interface. For example, an interface ISit-At-Able can represent an affordance sit-at-able. Its method void Sit(...) can be seen as a function.

For a software engineer, it is important to distinguish between a function and a construction. A lack of their differentiation can lead to a confusion in the
architectured software. There can actually be more software systems having a different construction decomposition yet sharing a functional decomposition. Design patterns are currently one of the most commonly used principle for sharing constructional elements among systems. Each design pattern is a constructional skeleton that can be applied to different functions.

In this section, we outlined a mapping of EE principles from τ-theory into SE. Our hypothesis is that a software system might be described by requirements on affordances, constructions, and functions. This information should be a base for an automatic creation of various software components. Thus, we consider the notion of affordances, constructions, and functions as a key element in our research.

2.3 Ontological Theories

The DELTA Theory (δ-theory) (Discrete Event in Linear Time Automaton) is about finite automatons. It defines so-called DELTA automaton explaining events, transition rules, occurrence of events, etc. DELTA automaton is an automaton applicable on a technical description of a system (e.g., a semaphore).

Without further explanation, finite automatons have definitely an important role in computer science. There are various types of automatons, pushdown automaton, Turing machine, etc. We would like to analyse the DELTA automaton deeper. We have to identify classes of problems where SE can benefit from DELTA automaton instead of other available formalisms.

Nevertheless, this theory does not seem to be a core of my doctoral thesis targeting a component software design. Thus, we do not put emphasis on δ-theory so far.

The PSI Theory (ψ-theory) (Performance in Social Interaction) deals with a communication and an interaction between subjects (human beings). Its generalised version, so-called General ψ-theory introduces a term transaction, and it defines a transaction axiom. It sees the organisation as a social system composed of actors. These actors interacts by performing coordination acts which are raised within patterns called transactions. As Dietz says: “Carrying through a transaction is a „game“ of entering into and complying with commitments” [15]. A regard of these commitments are products/services the subjects bring about in a coordination. This is actually an operating principle of General ψ-theory.

We believe that SE can benefit from the principle mentioned above. Especially, when dealing with requests on confirming various objects created in an application (e.g., entries in a database). Such a confirmation might involve more actors, one in a position of a creator and the second in a position of a confirmator of the created object. By mapping ψ-theory into these confirmation processes, we get a guarantee that all possible situations are covered. For example, a sit-
uation when a creator changes his mind shortly after asking for a confirmation (so-called revocation).

To prove our idea, we already mapped the principles of $\psi$-theory into so-called *confirmation engine*, and we integrated it in a commercial software CoRiMa. CoRiMa is a software developed by COPS GmbH using .NET. It is a multi-user client-server application and an application platform at once. It cooperates with an underlying risk-management system to that it communicates demands on financial-based data (e.g., deals, foreign exchanges, balances, etc.). Since much of this information is critical for the risk calculations, it cannot appear in the target risk-management systems unapproved by privileged users. COPS calls this integral principle *confirmation principle*. The unit supporting confirmation principle is the mentioned confirmation engine.

There are obvious similarities between transactions in $\psi$-theory and so-called confirmations in a confirmation principle. Thus, we based the implementation of a confirmation engine on findings of $\psi$-theory. Since CoRiMa is running at several corporate and banking environments across Europe, we believe that it is not just a toy example to prove a mapping of PSI Theory into software development. The corresponding article describing a confirmation engine was already accepted at CBI 2015 [6].

2.4 Technological Theories

As defined by Dietz: “Technological theories concerns understanding the design and making of things” [13]. Our research is focused on an improvement of a design of component-based applications. Thus, we believe the technological theories introduced by CIAO! might influence our work.

**The BETA Theory** ($\beta$-theory) (Binding Essence, Technology and Architecture). It is a theory about the design of systems as defined by $\tau$-theory. The $\beta$-theory becomes one of the most important theories in our research. It discusses mostly the notion of architecture, thus, it deals mainly with the architecturing of systems. The architecturing is a key notion in EE. Its view on the generic system development process is fully aligned to software engineering.

The $\beta$-theory covers important notions of distinguishing constructional and functional design. As already noted, a given set of functional requirements can have more constructions satisfying them. The theory explains that other engineering disciplines hold the same functional and constructional division. For example, there can be several differently looking houses yet based on the same set of functional requirement. This difference is explained by a substantial amount of a design freedom that architects may practise. When building a house, an architect can choose from various technologies and materials. The same is valid for SE. The software architect can choose from a substantial amount of technologies and components to build a software.
The next promising cornerstone for our research is Generic Requirements and Architecture Framework (GRAF). GRAF tries to answer a question of how the specific and generic requirements can be structured. It deals with the way of organising them in a manageable form. GRAF is formally defined as a tuple <S,D,A> [13]:

- S: a set of system kinds
- D: a set of design domains
- A: a set of areas of concern

We believe that by using those tuples we can build a precise component-based specification of a system.

Our current research of mapping β-theory into SE is outlined on the Figure 6. We would like to propose a way of architecturing software based on requirements defined more precisely and rigorously. We think the functional and constructional requirements can be described using GRAF. They contribute corresponding functional and constructional design domain. Each requirement can in addition define a specific characteristic, e.g., reliability, security, or efficiency.

We guess that the requirements may be enhanced by relevant ISO standards, e.g., ISO 25000 [1], which is a standard for System and Software Quality Requirements and Evaluation (SQuaRE).

We introduce so-called component repository. A range of components can be categorised according to functional and constructional requirements. We match the specific requirements with categories, and we choose the most appropriate component from a repository. The selected component is supplied with a necessary architecture knowledge and transformed into a necessary application (web, desktop, etc.). This transformation can benefit from findings on normalized systems or a code generation (e.g., GES[1]).

Needless to say that by software architecturing as described above, we still leave a room for a design freedom. The final application can still be customised and "coloured" by a developer.

3 Component-based Systems and TAO

In any programming ecosystem, a range of components can be identified. Frameworks and libraries offer an option to reuse available components. However, the common practise is to either reuse or reconfigure available component, or implement it new by fitting it in a required interface. The change of a component

---

1 GES is a Generative Engineering Studio developed by Codiscent. Its main focus is on a code generation using templates.
Fig. 6: Component-based design and an architecturing on different levels of an abstraction does not seem to be usual. It might increase the risk of incompatibilities, combinatorical effect, etc.

For instance, a framework Express on a platform NodeJS is a minimal and flexible web application framework. It gained a successful separation of concerns. HTML templates are independent on the rest of a framework, e.g., middleware or routing. One can choose from various modules during a development, e.g., a specific middleware module. However, one can only choose from modules on a certain abstraction level. The deployed system that is developed on a top of Express can be hardly transformed into a similar system just by changing templates. To achieve that, one would have to understand the entire deployed system. Thus, the replacebility of components is not governed, managed, and guaranteed.

It seems that a composition of a software from subsystems is nowadays quite well discussed. However, a transformation of a given complex system into another, yet similar, complex system seems to be a challenge. A rigorous and guaranteed approach is missing. It is not clear how to reuse maximum of modules of the old system, and how to exchange existing modules to meet requirements on a new system. In addition, it is not clear how to achieve a recursive module reuse. Thus, how to ensure that even systems originally not designed as reusable could be finally reused (at least from some extent).

The Figure 7 exhibits a functional decomposition of a system on the left. As discussed in τ-theory (TAO) [16], each node represents a substantially complex set of facts and rules. On the right, a corresponding constructional decomposition takes a place. A picture demonstrates that after slightly changing functional decomposition, the transformation of a corresponding constructional decomposition is not obvious.

We plan to develop a schema of annotations characterising components on different levels of an abstraction. Each annotation would transmit a meta-
information about a component. In terms of $\tau$-theory, the meta-information defines affordances of each component (e.g., a component dedicated for filtering of data defines, whether it is capable of data virtualisation or not).

The question is how to reuse the construction elements to be aligned with the change in functional decomposition (and with corresponding facts and rules). Our hypothesis is that a software system must have a well defined tree of a functional and constructional decomposition, and it must have proper annotations of each node. The research question is, which facts and rules we need, and how each node should be annotated to rigorously apply the required transformation.

4 Normalized Systems Overlap

Normalized systems are defined \[11\] as “Information systems that are stable with respect to a defined set of anticipated changes, which requires that a bounded set of those changes results in a bounded amount of impacts to the system”. Normalized systems generally deal with an evolvability of a software. Their effort is in preventing a combinatorical effect.

Our research is focused on reusability and creation of large components. We concentrate on reusability on a certain level of modularity and on a certain level of a detail. For example, let us assume we have a component capable of filtering data. For any reason, we cannot use that component in its full extent. Thus, based on the functional and constructional decomposition of that component, we have to identify its reusable parts and apply them on our requirements.

Our hypothesis is that normalized systems can help to improve characteristics of a reusability. They might support the quality of transformations as outlined in section 3. For example, if a given system will suffer from a combinatorical
effect, we might expect way more complicated transformation after its functional requirements change.

In addition, we have to analyse the evolvability of the original and transformed system. Without the proper handling of evolvability, the whole concept and research will be unusable in practice.

5 Future Work

We already proposed our vision of a component repository in section 2.4. Next mission is to design a feasible solution to realise that. We would like to combine the vision of a repository with a schema of annotations presented in section 3. The repository should be able to browse existing components and query them using annotations. Since the annotations are supposed to describe affordances of components, the repository should in fact allow searching of components based on their affordances. Thus, our next research will be focused on designing of a domain specific language (DSL) capable of annotating components.

The DSL must consider dependencies between components. We have to study theories about interfaces, e.g., contract-based programming [18] in Eiffel [19] or Haskell [2]. We have to inspect algebraic data types and formal specifications, e.g., Alloy specification language [12]. The DSL must reflect an appropriate comprehensive minimal set of annotations that need to be present for a system to be usable.

Next research question is: How to formalize mutual relation among and inside components, e.g., how to update one component when another updates. For this purpose we want to study functional-reactive programming and Elm language [4].

However, the obvious question is whether this extra work of formalizing design of systems, introducing annotations, and designing component repository would pay off. In our opinion, it will. Thanks to better design, we might bring better reusability and thus reduce costs on testing.

We will devise a proof of concept where we utilize our cooperation with COPS GmbH. We would like to introduce a component repository in CoRiMa and to integrate it with already implemented confirmation engine (see section 2.3). The domain of components in repository can be quite huge, e.g., front-end components, back-end components, etc. Thus, we will concentrate only on graphical components. We guess the complexity of a domain knowledge is in depth and breadth manageable in Ph.D. studies.
6 Conclusion

In this paper, we introduced our research focused on a mapping of EE theories into software engineering. We analysed each EE theory separately and proposed its mapping to SE (see section 2). The described mappings mostly concern only a surface of each theory. Thus, we would like to analyse the theories deeper and consider their broader influence on SE. Moreover, it is questionable whether the suggested mapping is applicable to all classes of IS, e.g., those suggested by C.W. Churchman [3]. Thus, we would like to inspect this topic further.

We think that the mappings of EE theories might significantly support component-based design. We outlined it in section 3, where we clarified the main topic of our research – transformations of component-based systems. We presented some aspects of that problem in terms of $\tau$-theory (TAO) that was discussed in section 2.2.

We showed how normalized systems might contribute our research in section 4, and we pointed out that findings of normalized systems can significantly influence the mentioned transformations.

In the end, we presented our intention to design a method that might ease transformations of component-based systems (see section 5), and we outlined our next research.

7 Acknowledgement

We would like to express thanks to COPS GmbH for an intensive cooperation on this research. This paper was written with the support of the SGS15 CTU grant no. 118/OHK3/1T/18.

References

11. Herwig Mannaert, J.V., Ven, K.: Towards evolvable software architectures based on systems theoretic stability (Jan 2011)
15. Jan, D., Jan, H.: Theories in Enterprise Engineering Memorandum - PSI
17. McIlroy, D.: Mass produced software components. NATO Conference on Software Engineering, Garmisch, Germany