Impact analysis approaches:

Source code: Many approaches investigate the impacts of changes by reasoning about inheritance relations, method-call behavior, and other dependencies between program entities. Source code files, class packages, classes, methods, statements, and variables are analyzed to predict the propagation of changes. However, such techniques are not applicable in the early phases of software design and requirements analysis, when no source code is available. Their rather technical nature also limits their application to programmers and is of little use for other stakeholders, such as project leaders. Static code analysis extracts facts from source code to build call graphs, slices, and other representations which are used to assess the impacts of a change. In contrast, dynamic and online approaches instrument the code or compiled binaries to collect information about method executions. These so called execution traces are either analyzed after program execution (dynamic) or on the fly (online) to enable the concurrent assessment of changes.

Architectural Models: Architectural models, such as UML component diagrams, enable the assessment of architectural changes on a more abstract level than source code. This enables impact analysis in earlier stages of development and in model based development (MBD), which has become more important in recent years. But dependent on the underlying modeling language, even architectural analysis allows for fine-grained results, for example when analyzing detailed UML class diagrams. Typical levels of granularity are systems, sub-systems, components, and classes.

Requirements Models: Formalized requirements are the first available artifacts during the software development process, and undergo many changes until the final version of a program has been implemented. If requirements are encoded in formal modeling languages such as UML, GRL or UCM, they can be subject of formal analysis as they adhere to a well-structured metamodel. In contrast, if they are expressed as plain text, only textual retrieval methods such as information retrieval (IR) can be applied.

Miscellaneous Artifacts: It is an acknowledged fact that changes do not only occur in source code or architectural models. Documentation, configuration files, bug trackers, and similar auxiliary content of software are also subject of frequent changes. However, changes to such entities can also affect the software, e.g. when a configuration file was changed. Performing impact analysis among such entities, mostly different types of files, has therefore become an issue to the research community as well.

Combined Scopes: The typical software development process is comprised of different phases, i.e. starting with requirements analysis and architectural reasoning, which are then followed by the implementation. When changing software, impacts do not limit themselves to certain kinds of artifacts. Changing a requirement can affect different architectural models, as well as already implemented source code components. Therefore, comprehensive analysis is required to trace impacts across all available artifacts. This however requires sophisticated concepts that are able to deal with a wide variety of possible types of artifacts.

Techniques of source code approaches:

Call graphs: Changed methods or procedures can affect other source code entities, which either call the m directly or indirectly. Analyzing the call-behavior of a system can therefore help to assess the impact of a method/procedure change. Thus, source code is analyzed statically while method calls are extracted and stored in a graph or matrix. This graph then enables developers to estimate the propagation of a given change. E.g: Chianti

Dependency analysis: There exist a variety of dependencies between source code entities, such as control, data or inheritance dependencies. They can be extracted by static source code analysis and either stored in a graph or a matrix. Based on dependencies between software artifacts, one can estimate the change propagation between them.

Program Slicing: Slicing is one application of static source code analysis and is build upon code dependency analysis. Slicing removes all program statements which are not related to the slicing criterion, i.e. which do not affect the state of a variable and thereby being of no use for impact analysis

Execution Traces: In contrast to static call graphs, dynamic execution traces contain only those methods which have been called during the execution of a program. Similar to call graphs, they allow to assess the impact of a method change by analyzing which methods were called after the changed method, thus being possibly impacted too. Approaches which rely on dynamic execution data were established to overcome the limitations of static slicing (expensive) and call graphs (imprecise).

Explicit Rules: Design, domain, and expert knowledge can be used to form strict impact rules, which determine which entities have to change if a certain entity changes. For example, if an interface is changed, all classes which implement this interface must be changed as well, dependent on the type of change (e.g. deletion of a method)

Information Retrieval: Information retrieval (IR) techniques exploit natural languages by searching for similar terms in different documents, to infer a relation between them

Probabilistic Models: Probabilistic models, such as Markow chains and Bayesian Belief Networks (BBN), allow to model the propagation of changes based on well explored mathematical models and theorems. Thus, they allow to compute the probability of an entity being impacted by a change.

History Mining: It mines clusters or patterns of entities from software repositories which often changed together, as a change to one entity of a cluster is likely to affect all the other entities within this cluster as well. E.g: For a given change in a software system, this technique computes that co-changes – what artifacts (such as files, classes, methods and lines) should also change. This technique can also identify non-source code artifacts that are frequently modified with a given change. One advantage of this method is it requires on stable software change history or bug repositories, which might not be available.

Combined Technologies: Combining MSR which analyzes a series of versions with traditional impact analysis. which is focused on a single version could yield several advantages. Traditional analysis applied on single versions of a program is not adaptive, whereas MSR-based approaches rely on historical data, which might be incomplete or outdated and therefore inferring wrong impacts. Combining both methodologies to improve the prediction of changes. The basic idea is to apply traditional techniques on single revisions and to infer additional dependencies from version histories using MSR approaches. Using the achieved results to cross validate each other is a potential technique to further improve impact analysis. Further refine the idea by proposing to convert the source code into an XML representation using srcML to enable source code dependency analysis. The authors plan to use the mining tool sqminer and the diff-tool codeDiff to mine fine-grained co-changes at code level. The last step is focused on matching the granularity of entities obtained through MSR and traditional analysis to incorporate both results by either forming the union (Disjunctive Approach) or intersection (Conjunctive Approach) of both sets.

Techniques of Architectural Models:

Techniques of Requirement Models: