(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
1 December 2016 (01.12.2016)

WIPO | PCT

(10) International Publication Number
WO 2016/190876 A1

(51) International Patent Classification:
G06F 17/30 (2006.01) *G06F 17/00* (2006.01)

(21) International Application Number:
PCT/US2015/032923

(22) International Filing Date:
28 May 2015 (28.05.2015)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant: **HEWLETT PACKARD ENTERPRISE DEVELOPMENT LP** [US/US]; 11445 Compaq Center Drive West, Houston, TX 77070 (US).

(72) Inventors: **KOGAN, Olga**; Hewlett-Packard Company, Shabazi 26, 56100 Yehud (IL). **LEVIN, Amit**; Hewlett-Packard Company, Altalef St No. 5, Shabazi St. No. 26, 56100 Yehud (IL).

(74) Agents: **GARDINER, Austin W.** et al.; Hewlett Packard Enterprise, 3404 E. Harmony Road, Mail Stop 79, Fort Collins, CO 80528 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to the identity of the inventor (Rule 4.17(i))*

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

Published:

— *with international search report (Art. 21(3))*

(54) Title: DEPENDENCY RANK BASED ON COMMIT HISTORY

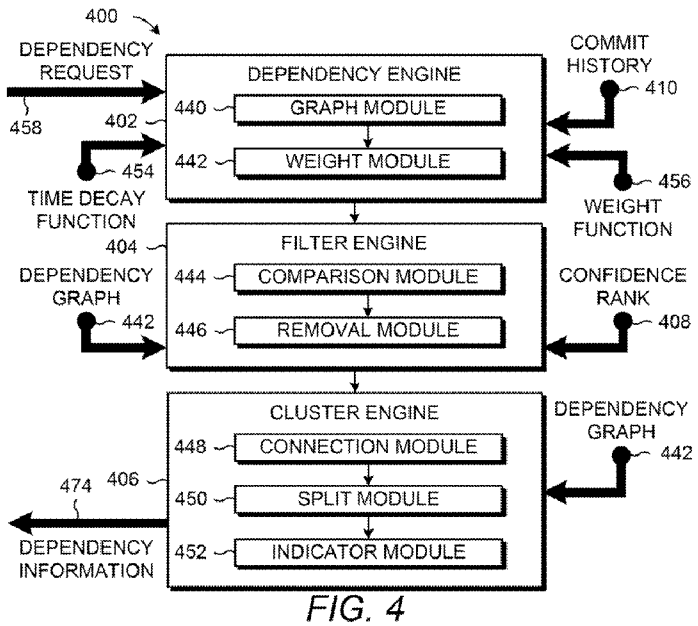

FIG. 4

(57) Abstract: In one example implementation, an example system includes a dependency engine, a filter engine, and a cluster engine. In the one example implementation, a file pair of a first commit log entry of a plurality of commit log entries in a commit history is identified by the dependency engine. In the one example implementation, a dependency rank is assigned to the file pair by the dependency engine based on a number of times the first file of the file pair and the second file of the file pair appear together in the plurality of commit log entries. In the one example implementation, a dependency rank is compared to a confidence rank by the filter engine. In the one example implementation, a second file is indicated as dependent on a first file when the dependency rank of the file pair achieves the confidence rank.

WO 2016/190876 A1

Dependency Rank based on Commit History

BACKGROUND

[0001] Software development of code projects may demand organization of files and management software of the source code may be used to manage the code as it changes over time. For example, a version control system (e.g., a source control tool) may refer to changes in the source code to a repository where developers may update a latest version of code and allow multiple developers to work on a project concurrently.

BRIEF DESCRIPTION OF THE DRAWINGS

[0002] Figures 1 and 2 are block diagrams depicting example dependency identification systems.

[0003] Figure 3 depicts an example environment in which various dependency identification systems may be implemented.

[0004] Figure 4 depicts example interaction by example components used to implement an example dependency identification system.

[0005] Figures 5 and 6 are flow diagrams depicting example methods for finding a dependency of files in a project.

[0006] Figure 7 depicts an example commit history and an example dependency list based on the example commit history.

[0007] Figure 8 depicts an example commit history and example dependency graphs based on the example commit history.

DETAILED DESCRIPTION

[0008] In the following description and figures, some example implementations of dependency identification apparatus, dependency identification systems, and/or methods for finding a dependency of files in a project are described. Software

development projects may be managed with a lack of understanding of design and dependencies, such as when each developer of the project knows the particular dependencies, but that information is not always communicated to other developers. Thus, the design information and the dependency information may be scattered across developers of a process. As files of the project are revised over time, commonly by multiple developers, the complexity of the source code for a project may increase as well as the effort in maintaining familiarity with the code. To understand dependencies in the source code, a developer may spend time reviewing the code, relying on documentation produced by other developers, executing static code analysis tools to identify the dependency information, or some combination thereof..

[0009] Various examples described below relate to analyzing the commit history to identify dependency information based on developer's commits. By identifying an association of files based on files committed together for a commit and using a number of distinct developers, a dependency among files may be identified. As described herein, a dependency rank may be associated with a pair of files and based on the level of the dependency rank, then the files may be identified as dependent on each other. For example, if the dependency rank achieves a value with which confidence may be had in their dependency, which discussed herein as a comparison with a confidence rank. Errors based on a single developer's coding strategy may be subsided by utilizing the number of distinct developers committing the same file pair. As used herein, a file pair is a representation to identify a group of files as distinct from another group of files.

[0010] The terms "include," "have," and variations thereof, as used herein, mean the same as the term "comprise" or appropriate variation thereof. Furthermore, the term "based on," as used herein, means "based at least in part on." Thus, a feature that is described as based on some stimulus may be based only on the stimulus or a combination of stimuli including the stimulus.

[0011] Figures 1 and 2 are block diagrams depicting example dependency identification systems 100 and 200. Referring to Figure 1, the example dependency identification system 100 of Figure 1 generally includes a dependency engine 102, a filter engine 104, and a cluster engine 106. In general, a group of files may be identified as dependent on each other by a cluster engine 106 where the dependency engine 102

associates a dependency rank with the files that are committed together and the filter engine 104 filters files based on the dependency rank using a confidence rank 108. The example commit history 110 is simplified in Figure 1 and generally would contain a plurality of commit log entries, such as commit log entry 112.

[0012] The dependency engine 102 represents any circuitry or combination of circuitry and executable instructions to associate a dependency rank with files that are committed together in a commit history. The dependency rank has a relationship with the number of times a pair of files appear together in the commit history. The dependency engine 102 may take into account the diversity of the developers performing the commits. An entity (e.g., a developer) that performs a commit is discussed as a "committer" herein. For example, the dependency engine 102 may represent a combination of circuitry and executable instructions to identify a file pair of a first commit log entry of a plurality of commit log entries in a commit history and assign a dependency rank to the file pair based on a number of times a first file of the file pair and a second file of the file pair appear together in the plurality of commit log entries and a number of distinct committers that commit the first file and the second file together.

[0013] The file pairs may be identified by the dependency engine 102 by, for example, making a file pair for each two-file combination of the plurality of files listed in a commit log entry. A commit log entry, as used herein, represents a data structure containing information of a commit. For example, the commit log entry may be a data structure that includes fields for a committer identifier and a plurality of files. For another example, the commit log entry may be a data structure comprising fields for a commit identifier, a committer identifier, a commit time, and a list of files of the commit. As used herein, a commit identifier represents a number, a character, a string, a label, or other distinctive value that represents a commit log entry and a committer identifier represents a number, a character, a string, a label, or other distinctive value that represents the entity performing the commit, such as a name of the committer or an employee identification number of the committer.

[0014] A dependency rank, as used herein, represents a number, a character, a string, a category, a label, or any other identifier that represents a comparable value

associated with the number of times a file is committed with another file. As mentioned above, the dependency rank is assigned based on the number of commits that contain the file pair being assigned the dependency rank and the number of distinct committers that perform the number of commits. The weight given to those factors may be adjusted according to the project and/or attributes of the commit history. A weight function may be used to determine the dependency rank. For example, the dependency engine 102 may determine the dependency rank as a weighted value based on a weight function using a set of parameters comprising a number of commits, a number of distinct committers that perform the commits, and a time decay function (e.g., where the time decay function emphasizes a first commit that is committed more recently than a second commit with a greater weighted value than the second commit). The weight function may perform weighting of the dependency rank (e.g., weight parameters of the weight function) based on the diversity of the committers and/or the diversity of the commit times. For example, the weight function may at least one of weight the dependency rank based on a number of distinct committers identified by a plurality of commit log entries that include the first file and the second file and/or weight the dependency rank based on a comparison of a time window to a commit time of the commit log entry.

[0015] The dependency engine 102 may produce a dependency list based on the identified file pairs and the dependency ranks assigned to the file pairs. The dependency list may be used to identify a file that depends on another file while taking into consideration the dependency rank. As an example, Figure 7 depicts an example dependency list 780 formed by a dependency identification system, such as dependency identification system 100, using the data of the commit history 710 where the file pairs are listed in order of dependency rank and a file or a file pair may be searched for in the dependency list.

[0016] The dependency engine 102 may represent circuitry or a combination of circuitry and executable instructions to manage a dependency graph based on a commit history, where the dependency graph is usable, for example, to identify a file pair and assign a dependency rank to the file pair. The dependency graph may comprise nodes and edges where each node represents a file of the commit history and each edge

represents a file pair and is associated with a dependency rank correlated with the file pair. For example, the dependency engine 102 may represent a combination of circuitry and executable instructions to create a dependency graph comprising a first node that represents the first file of a first plurality of files of a first commit log entry in a commit history, create a first edge between the first node and a second node that represents a second file of the first plurality of files of the first commit log entry, and increase a dependency rank of the first edge when a second commit log entry of the plurality of commit log entries contains a second plurality of files comprising the first file and the second file. Referring to Figure 8, a commit history 810 may be used by the system 100 to form a graph 882 that represents files of a plurality of commit log entries of the commit history 810 as nodes and file pairs of the plurality of commit log entries as edges.

[0017] The dependency engine 102 may represent circuitry or a combination of circuitry and executable instructions to examine the files of a commit log entry and identify a related code in the files of the commit log entry. For example, the dependency engine 102 may represent a combination of circuitry and executable instructions to retrieve code of files of the commit log entry, identify a code pair based on changes to a first file and a second file in the commit log entry, determine a regularity of the code pair based on a number of times the code pair appears in the commit history, and determine the dependency rank associated with the file pair based on the regularity. A code pair, as used herein, refers to a pair of sections of code that are changed in the files of the file pair compared to the previously committed versions of the files. For example, a first function of a first file may be changed in a commit where a second function is changed of a second file. Thus, the dependency engine 102, may for example, identify a code pair when a first section of code changes in the first file and a second section of code changes in the second file, determine a regularity of the code pair based on a number of times the first section of code and the second section of code change together in the commit history, and modify a dependency rank of the file pair using the regularity (e.g. increase the dependency rank when the regularity is relatively high or decrease the dependency rank when the regularity is relatively low). As used herein, a regularity represents a number, a character, a string, a label, or other comparable value that

indicates an amount of times the code pair changes together in the commit history (e.g., a rate that a first section of code in a first file of the project changes with a second section of code in a second file). The code pairs may be used as part of the dependency information, such as in building the dependency graph. For example, by using code pair information, the dependency ranks may be associated with a pair of file and function combinations where the nodes in the dependency graph represent a combination of a file and a section of code in the file and the edges represent a file pair/code pair with second combination of a file and a section of code. In this manner, a greater granularity of dependency may be identified because the analysis of the files committed together are analyzed deeper.

[0018] The filter engine 104 represents any circuitry or combination of circuitry and executable instructions to filter a dependency rank based on a confidence rank. For example, the filter engine 104 may represent a combination of circuitry and executable instructions to compare the dependency rank of the file pair to a confidence rank and disregard any file pairs that do not achieve the confidence rank. As used herein, a confidence rank is a number, a character, a string, a label, or other value to represent a threshold of confidence that the dependence rank indicates an actual dependency. For example, if a file pair is committed only once in the commit history then there would be a low confidence in that files of the file pair are indeed dependent on each other and might instead, have been committed together merely based on the method of development by a particular developer. In this manner, the filter engine 104 may, for example, identify file pairs that have a sufficient reliability of being dependent on each file of the file pair.

[0019] The filter engine 104 may, for example, represent circuitry or a combination of circuitry and executable instructions to identify unreliable file pairs using the confidence rank and removes the file pairs from the possible results of the system 100 or otherwise indicates the unreliability the file pair as having a dependency. For example, the filter engine 104 may represent a combination of circuitry and executable instructions to flag or otherwise indicate that a file pair does not achieve a confidence rank. For another example, the filter engine 104 may represent a combination of circuitry and executable instructions to remove the first edge from the dependency

graph when the dependency rank is below a confidence rank. For yet another example, the filter engine 104 may represent a combination of circuitry and executable instructions to change a visual characteristic of a node and/or edge associated with the file pairs that do not achieve the confidence rank. Referring to Figure 7, a divider 712 is shown as a representation of a "cut off" point of confidence of file pairs actually having a dependency where the division is at the dependency rank of five. Thus, in that example, the file pairs having a dependency rank of five or above achieve the confidence rank. Referring to Figure 8, a filter engine, such as filter engine 104, may form the graphs of 884 from the graph 882 by removing edges that do not achieve a confidence rank. In the example of Figure 8, the edges having a dependency rank of four have been removed. Edge removal is an example form of a visual indication that an edge does not achieve a confidence rank. Other example visual indication include a change in the opacity, a change in color, an addition of an icon, an addition of a divider, etc.

[0020] A confidence rank may be set by the system 100, by a user or administrator of the system 100, by a machine learning technique, or some other way to identify a division between file pairs that are more likely to have an actual code dependency than file pairs who may be merely committed at the same time. The confidence rank may be based on a project attribute of a project associated with the commit history, such as a number of commit log entries in the commit history. As used herein, a project attribute may be any characterization of the project and may represent (and/or incorporate) a plurality of project attributes. For example, the project attribute may be at least one of a project size, a number of total commits of the project, a number of changes of a section of the project, and/or a number of developers of the project.

[0021] The cluster engine 106 represents any circuitry or combination of circuitry and executable instructions to indicate the second file is dependent on the first file when the first file pair achieves the confidence rank. For example, the cluster engine 106 may represent a combination of circuitry and executable instructions to identify a cluster of nodes that are connected in the dependency graph and cause the cluster of nodes to display. In the example of file pair removal by the filter engine 104, the cluster engine 106 may indicate the file dependencies by providing the file pairs that achieve the confidence rank, such as in a dependency list or a dependency graph. Dependency

and/or the degree of dependency (as indicated by the, for example, the relative differences in dependency ranks) among files may be indicated, for example, by an identifier, an icon, a font or font style, a color, an opacity, an edge of a graph, a distance between nodes in a graph, or other visual indication.

[0022] In the example of providing a dependency graph, the cluster engine 106 may represent any circuitry or combination of circuitry and executable instructions to identify nodes of a cluster that are not a file pair in the commit history. For example, the cluster engine 106 may identify a cluster of nodes in the dependency graph based on edges of nodes of the cluster and identify a third node that lacks a direct edge to a first node when the cluster of nodes includes a first node, a second node, and a third node.

[0023] When the cluster engine 106 identifies the indirect relationship between files (e.g., the nodes of the files are in the same cluster but are not directly connected by an edge in the graph representation), the cluster engine 106 may identify that the cluster of nodes represents a plurality of dependency clusters and should be separated into a plurality of sub-graphs. The cluster engine 106 may represent circuitry or a combination of circuitry and executable instructions to split the cluster of nodes into a plurality of sub-graphs. For example, the cluster engine 106 may represent a combination of circuitry and executable instructions to separate the cluster into a plurality of sub-graphs based on nodes of the cluster that do not have a direct connection where some nodes in the dependency graph may be duplicated.

[0024] Because nodes may be shared among the final sub-graphs, the clusters having shared nodes may be duplicated. For example, a cluster may be duplicated wherein the shared nodes among a first sub-graph of the plurality of sub-graphs and a second sub-graph of the plurality of sub-graphs are duplicate graphs. The cluster engine 106 may trim the nodes and edges of the duplicate graphs to create different sub-graphs. For example, the cluster engine 106 may remove a first node representing a first file of a file pair from a first duplicate sub-graph and remove a second node representing a second file of a file pair from a second duplicate sub-graph where the resulting sub-graphs include a first sub-graph of the plurality of sub-graphs that comprises the first node, the second node, and the first edge connecting the first node and second node and a second sub-graph of the plurality of sub-graphs that comprising

the second node, the third node, and a second edge connecting the second node and the third node. Referring to Figure 8, the graphs 886 represent the sub-graphs of the filtered dependency graph of 884, where the nodes representing files E and H have been duplicated and the nodes G and F are removed from one graph (due to no direct edge with node D) and the node D removed from the other graph. For another example, sub-graphs may indicate architecture cross-layer dependencies such as in detecting a cluster of application programming interface (API) functionalities that depend on changes in external and internal API code. By identifying dependencies and narrowing the dependencies, confidence may be had in the results of the dependency identification system 100. For example, the dependency identification system 100 may take into account not just an overall number of times a file pair is committed in a commit history, but also the number of distinct committers, a time window, and code pair information (e.g., the regularity of a code pair committed in the commit history) to determine a dependency rank when indicating a dependency based on a comparison of a dependency rank to a confidence rank as described herein.

[0025] In some examples, functionalities described herein in relation to any of Figures 1-3 may be provided in combination with functionalities described herein in relation to any of Figures 4-8.

[0026] Figure 2 depicts the example system 200 may comprise a memory resource 220 operatively coupled to a processor resource 222. Referring to Figure 2, the memory resource 220 may contain a set of instructions that are executable by the processor resource 222 as well as any data used by the system 200. For example, the memory resource 220 may contain a set of instructions, a confidence rank 208, and a commit history 210. The set of instructions are operable to cause the processor resource 222 to perform operations of the system 200 when the set of instructions are executed by the processor resource 222. The set of instructions stored on the memory resource 220 may be represented as a dependency module 202, a filter module 204, and a cluster module 206. The dependency module 202, the filter module 204, and the cluster module 206 represent program instructions that when executed function as the dependency engine 102, the filter engine 104, and the cluster engine 106 of Figure 1, respectively. The processor resource 222 may carry out a set of instructions to execute

the modules 202, 204, 206, and/or any other appropriate operations among and/or associated with the modules of the system 200. For example, the processor resource 222 may carry out a set of instructions to modify a dependency rank of an edge of a dependency graph when a commit log entry contains a first file and a second file and the first file is represented by a first node of the dependency graph connected by the edge to a second node of the dependency graph representing the second file, compare the dependency rank to a confidence rank, and indicate the first file and the second file have a dependency when the dependency rank achieves the confidence rank. For another example, the processor resource 222 may create a dependency graph by carrying out a set of instructions to add a first node and a second node to the dependency graph when the commit log entry contains a plurality of files that includes the first file and the second file (and the dependency graph does not already include the first node and the second node) and create the edge between the first node and the second node.

[0027] Although these particular modules and various other modules are illustrated and discussed in relation to Figure 2 and other example implementations, other combinations or sub-combinations of modules may be included within other implementations. Said differently, although the modules illustrated in Figure 2 and discussed in other example implementations perform specific functionalities in the examples discussed herein, these and other functionalities may be accomplished, implemented, or realized at different modules or at combinations of modules. For example, two or more modules illustrated and/or discussed as separate may be combined into a module that performs the functionalities discussed in relation to the two modules. As another example, functionalities performed at one module as discussed in relation to these examples may be performed at a different module or different modules. Figure 4 depicts yet another example of how functionality may be organized into engines and/or modules.

[0028] The processor resource 222 is any appropriate circuitry capable of processing (e.g., computing) instructions, such as one or multiple processing elements capable of retrieving instructions from the memory resource 220 and executing those instructions. For example, the processor resource 222 may be a central processing unit

(CPU) that enables finding a dependency of files in a project by fetching, decoding, and executing modules 202, 204, and 206. Example processor resources include at least one CPU, a semiconductor-based microprocessor, a programmable logic device (PLD), and the like. Example PLDs include an application specific integrated circuit (ASIC), a field-programmable gate array (FPGA), a programmable array logic (PAL), a complex programmable logic device (CPLD), and an erasable programmable logic device (EPLD). The processor resource 222 may include multiple processing elements that are integrated in a single device or distributed across devices. The processor resource 222 may process the instructions serially, concurrently, or in partial concurrence.

[0029] The memory resource 220 represents a medium to store data utilized and/or produced by the system 200. The medium is any non-transitory medium or combination of non-transitory media able to electronically store data, such as modules of the system 200 and/or data used by the system 200. For example, the medium may be a storage medium, which is distinct from a transitory transmission medium, such as a signal. The medium may be machine-readable, such as computer-readable. The medium may be an electronic, magnetic, optical, or other physical storage device that is capable of containing (i.e., storing) executable instructions. The memory resource 220 may be said to store program instructions that when executed by the processor resource 222 cause the processor resource 222 to implement functionality of the system 200 of Figure 2. The memory resource 220 may be integrated in the same device as the processor resource 222 or it may be separate but accessible to that device and the processor resource 222. The memory resource 220 may be distributed across devices.

[0030] In the discussion herein, the engines 102, 104, and 106 of Figure 1 and the modules 202, 204, and 206 of Figure 2 have been described as circuitry or a combination of circuitry and executable instructions. Such components may be implemented in a number of fashions. Looking at Figure 2, the executable instructions may be processor-executable instructions, such as program instructions, stored on the memory resource 220, which is a tangible, non-transitory computer-readable storage medium, and the circuitry may be electronic circuitry, such as processor resource 222, for executing those instructions. The instructions residing on the memory resource 220

may comprise any set of instructions to be executed directly (such as machine code) or indirectly (such as a script) by the processor resource 222.

[0031] In some examples, the system 200 may include the executable instructions may be part of an installation package that when installed may be executed by the processor resource 222 to perform operations of the system 200, such as methods described with regards to Figures 4-8. In that example, the memory resource 220 may be a portable medium such as a compact disc, a digital video disc, a flash drive, or memory maintained by a computer device, such as a service device 334 of Figure 3, from which the installation package may be downloaded and installed. In another example, the executable instructions may be part of an application or applications already installed. The memory resource 220 may be a non-volatile memory resource such as read only memory (ROM), a volatile memory resource such as random access memory (RAM), a storage device, or a combination thereof. Example forms of a memory resource 220 include static RAM (SRAM), dynamic RAM (DRAM), electrically erasable programmable ROM (EEPROM), flash memory, or the like. The memory resource 220 may include integrated memory such as a hard drive (HD), a solid state drive (SSD), or an optical drive.

[0032] Figure 3 depicts example environments in which various example dependency identification systems may be implemented. The example environment 390 is shown to include an example system 300 for finding a dependency of files in a project. The system 300 (described herein with respect to Figures 1 and 2) may represent generally any circuitry or combination of circuitry and executable instructions to find a dependency of files in a project. The system 300 may include a dependency engine 302, a filter engine 304, and a cluster engine 306 that are the same as the dependency engine 102, the filter engine 104, and the cluster engine 106 of Figure 1, respectively, and the associated descriptions are not repeated for brevity. As shown in Figure 3, the engines 302, 304, and 306 may be integrated into a compute device, such as a service device 334. The engines 302, 304, and 306 may be integrated via circuitry or as installed instructions into a memory resource of the compute device.

[0033] The example environment 390 may include compute devices, such as developer devices 332, service devices 334, and user devices 336. A first set of

instructions may be developed and/or modified on a developer device 332. For example, an application may be developed and modified on a developer device 332, committed to a repository via a source control tool 344, and stored onto a repository server, such as a service device 334. The service devices 334 represent generally any compute devices to respond to a network request received from a user device 336, whether virtual or real. For example, the service device 334 may operate a combination of circuitry and executable instructions to provide a network packet in response to a request for data, a page, or functionality of an application. The user devices 336 represent generally any compute devices to communicate a network request and receive and/or process the corresponding responses. For example, a browser application may be installed on the user device 336 to receive the network packet from the service device 334 and utilize the payload of the packet to display an element of a page via the browser application.

[0034] The compute devices may be located on separate networks 330 or part of the same network 330. The example environment 390 may include any appropriate number of networks 330 and any number of the networks 330 may include a cloud compute environment. A cloud compute environment may include a virtual shared pool of compute resources. For example, networks 330 may be distributed networks comprising virtual computing resources. Any appropriate combination of the system 300 and compute devices may be a virtual instance of a resource of a virtual shared pool of resources. The engines and/or modules of the system 300 herein may reside and/or execute “on the cloud” (e.g., reside and/or execute on a virtual shared pool of resources).

[0035] A link 338 generally represents one or a combination of a cable, wireless connection, fiber optic connection, or remote connections via a telecommunications link, an infrared link, a radio frequency link, or any other connectors of systems that provide electronic communication. The link 338 may include, at least in part, intranet, the Internet, or a combination of both. The link 338 may also include intermediate proxies, routers, switches, load balancers, and the like.

[0036] The data store 340 may contain information utilized by the engines 302, 304, and 306. For example, the data store 340 may store a confidence rank 308 and a dependency graph 342.

[0037] Referring to Figures 1-3, the engines 102, 104, and 106 of Figure 1 and/or the modules 202, 204, and 206 of Figure 2 may be distributed across devices 332, 334, 336, or a combination thereof. The engine and/or modules may complete or assist completion of operations performed in describing another engine and/or module. For example, the dependency engine 302 of Figure 3 may request, complete, or perform the methods or operations described with the dependency engine 102 of Figure 1 as well as the filter engine 104 and the cluster engine 106 of Figure 1. Thus, although the various engines and modules are shown as separate engines in Figures 1 and 2, in other implementations, the functionality of multiple engines and/or modules may be implemented as a single engine and/or module or divided in a variety of engines and/or modules. In some example, the engines of the system 300 may perform example methods described in connection with Figures 4-8.

[0038] Figure 4 depicts example interaction by example components used to implement an example dependency identification system 400. Referring to Figure 4, the example engines of Figure 4 generally include a dependency engine 402, a filter engine 404, and a cluster engine 406 that may represent the same engines as the dependency engine 102, the filter engine 104, and the cluster engine 106 of Figure 1, respectively. The example engines/modules of Figure 4 may be implemented on a compute device, such as a service device 334 of Figure 3.

[0039] The example system 400 of Figure 4 is initiated with a dependency request 450. The example dependency engine 402 uses program instructions, such as a graph module 440 and a weight module 442, to facilitate identification of file pairs in the commit history and assigning a weighted dependency rank to each file pair. The dependency engine 402 of Figure 4 utilizes a weight function 456 and a time decay function 454 when executing the weight module 442 to cause a processor resource to assign a dependency rank to a file pair. The dependency engine 402 of Figure 4 executes the graph module 456 to cause a processor resource to create and/or update a dependency graph 442, which is filtered by the filter engine 404 using the confidence

rank 408. The filter engine 442 executes the comparison module 444 to cause a processor resource to perform the comparison of a dependency rank to the confidence rank 408. The filter engine 442 executes the removal module 446 to remove any edges of the dependency graph 442 that fail to achieve the confidence rank 408.

[0040] The example system 400 provides the filtered dependency graph 442 to the cluster engine 406 to identify a cluster of files that are dependent on each other. The cluster engine 406 of Figure 4 utilizes program instructions, such as a connection module 448, a split module 450, and an indicator module 452, to facilitate indication of the dependencies in the commit history 410. For example, the cluster engine 406 executes the connection module 448 to cause a processor resource to identify clusters of the dependency graph 442 that have nodes that are not directly connected by an edge, executes the split module 450 to cause a processor resource to split the identified clusters by duplicating the identified clusters and removing nodes and edges associated with nodes that are not directly connected, and executes the indicator module 452 to visually indicate the nodes that have connections representing the likely dependency relationships of the files of the project using a visual indicator as discussed herein. The resulting dependency information 474 is then provided by the cluster engine 408.

[0041] Figures 5 and 6 are flow diagrams depicting example methods for finding a dependency of files in a project. Referring to Figure 5, example methods for finding a dependency of files in a project may generally comprise building a dependency graph, filtering the dependency graph using a confidence rank, creating a plurality of sub-graphs based on the filtered dependency graph, and marking a file of a sub-graph as having a dependency. The example method of Figure 5 generally comprises operations performable by a dependency identification system, such as dependency identification system 100, as described herein.

[0042] At block 502, a dependency graph is built having a plurality of nodes that represent a plurality of files in a commit history. As described herein, the dependency graph includes a plurality of edges among the plurality of nodes based on whether the plurality of files represented by the plurality of nodes are contained in commit log entries of the commit history. The edges are associated with dependency ranks, such as dependency ranks weighted based on the number of times a file pair is committed to the

commit history by a distinct committer, where the same files committed by a separate committer indicates a relationship between the files rather than development strategy, for example. The dependency graph may be built by a dependency engine, such as the dependency engine 102 of Figure 1.

[0043] At block 504, the dependency graph is filtered using a confidence rank. As described herein, a filter engine, such as the filter engine 104 of Figure 1, may remove edges from the dependency graph that do not achieve the confidence rank or otherwise indicate there may be a lack of confidence in a dependency of a file pair represented by an edge. At block 506, a plurality of sub-graphs are created based on the filtered dependency graph. For example, large clusters of nodes may be split into smaller sub-graphs to narrow the dependencies to be more direct. This may be done via a cluster engine, such as a cluster engine 106 of Figure 1, by removing nodes of a cluster that are indirectly connected by two or more edges (where the number of edges of a path between nodes representing a file pair indicates an indirect relationship and the greater number of edges indicates a greater degree of indirectness of the relationship).

[0044] At block 508, a file of a sub-graph is marked as having dependency on a complementary set of files in the sub-graph. As used herein, a complementary set of files in sub-graph represents the other files of the sub-graph with regard to a particular file in the sub-graph. In this manner, the each file of a sub-graph may be marked as being dependent on the other files of the sub-graph. As discussed herein, a cluster engine, such as the cluster engine 106 of Figure 1, may perform the marking of a file or otherwise indicate a dependency relationship among a files, which indication may, for example, be visual or indicated in the data of a dependency identification system, such a flag set in a dependency list of the dependency identification system 100 of Figure 1.

[0045] Figure 6 includes blocks similar to blocks of Figure 5 and provides additional blocks and details. In particular, Figure 6 depicts additional blocks regarding providing dependency information and details generally regarding blocks 502, 504, 506, and 508. The example method of Figure 6 generally comprises operations performable by a dependency identification system, such as dependency identification system 100, as described herein.

[0046] As indicated at block 602 of Figure 6, a dependency graph may be built by creating nodes, connecting nodes with edges, and associating a dependency rank with the edges. At block 604, a node is created in the dependency graph that represents a file of the commit history. For example, a first node of a plurality of nodes is created when the first node does not exist in the dependency graph and the commit log entry contains the first file to be represented by the first node. At block 606, nodes are connected with an edge when the associated files are both in a commit log entry. For example, a first node is connected via a first edge of the dependency graph to a second node of a plurality of nodes of the dependency graph by creating the first edge between the nodes when the first edge does not exist in the dependency graph and the commit log entry comprises the second file that is represented by the second node. At block 608, a dependency rank is associated with the first edge based on a number of distinct committers and a number of commit log entries that include the files associated with the edge, such as the first file and the second file connected by the first edge in the previous example.

[0047] As indicated at block 610, a dependency graph may be filtered using a confidence rank by identifying a project attribute and determining a confidence rank based on the project attribute. At block 612, a project attribute is identified, for example, by a filter engine, such as the filter engine 104 of Figure 1. The project attribute, as discussed herein, may be any appropriate property of the project that may affect a confidence level in a dependency between files. For example, a filter engine may identify at least one of a project size, a number of total commits of the project, and/or a number of committers of the project, where these project attributes may indicate the number of files being worked with which would reflect on the confidence level of any individual commit log entry. A number of file pair appearances in a commit history may seem large for a small project size and may seem small for a large project size. Therefore, the characteristics of the project are to be taken into account when determining the confidence rank as indicated at block 614. For example, the confidence rank may be determined based on at least one of the project size, the number of total commits, and the number of committers. The confidence rank may be determined using

a weighted function, a machine learning technique, or otherwise adjusted as the project attributes change.

[0048] As indicated at block 616, a plurality of sub-graphs may be created by identifying a plurality of clusters of nodes, duplicating a cluster, and removing a node from the duplicate cluster. At block 618, a plurality of clusters of nodes are identified based on indirect connections in the cluster. For example, a first cluster may be identified where the first cluster of the plurality of clusters is a first group of nodes that do not connect to a second group of nodes, such as after edges of the dependency graph are filtered out for not satisfying the confidence rank determined at block 614. An indirect connection may indicate that a cluster may contain multiple dependency clusters within the cluster. At block 620, a cluster is duplicated when the cluster contains nodes that are not connected by an edge. For example, a first cluster may be duplicated when the first cluster contains a first node and a second node and there is not an edge between the first node and the second node. At block 622, nodes (and associated edges) are removed from the duplicated clusters. Sub-graphs that are smaller than the duplicated cluster are extracted from the each duplicated cluster by creating sub-graphs based on nodes that are identified as not being directly connected in the duplicated cluster. Indirectly connected nodes identified at block 618 are removed separately from the duplicated clusters so that a plurality of sub-graphs are tailored to contain different nodes and/or edges. With reference to Figure 8 for example, the nodes representing files E and H have been duplicated and the nodes G and F are removed from one graph (due to no direct edge with node D) and the node D removed from the other graph. For another example, a first node is removed from a first duplicate of a first cluster and a second node is removed from a second duplicate of the first cluster, where the first node and second node are not directly connected in the filtered dependency graph resulting from block 610.

[0049] The files of the sub-graphs formed at block 616 are accordingly marked as having dependencies on the other files in the sub-graphs at block 624. As discussed above, the marking may be made on the data representation and/or visually presented. The resulting dependency information may be provided at block 626, where the dependency information comprises the files of a file pair that are likely to be dependent

on each other. Examples of providing dependency information include at least one of providing a list of dependency clusters in the project based on the plurality of sub-graphs, providing metric information regarding the first sub-graph, providing a list (e.g., a checklist) of dependency files of the first sub-graph associated with a checked-out file retrieved by a developer, and providing a visual representation of the plurality of sub-graphs via a web page (e.g., displaying a filtered dependency graph with dependencies indicated by edges between nodes on a browser application of a client device 336 of Figure 3). For example, a checklist of configuration file dependencies within a commit history retrieved from a source control tool may be provided to a developer (e.g., a build manager) on an open source project that is working on supporting functionality associated with the set of configuration files by using the example method of Figure 6.

[0050] Figures 7 and 8 depict example commit histories and results of identifying dependencies based on example dependency identification systems and/or example methods for finding a dependency of files in a project as described herein. Figures 7 and 8 have been described with reference to the descriptions of Figure 1, but may exemplify the example systems and/or methods described with reference to Figures 2-6 as well. Example uses for the example systems and example methods described herein include discovery of configuration file dependencies, architecture cross-layer dependencies, feature dependencies, deployment dependencies, etc.

[0051] For example in a first example project, the example systems and/or example methods described herein may detect a cluster of the build configuration files (e.g., pom.xml) when each time a new version is released, a plurality of build files are modified with the release version.

[0052] For another example in a second example project, the example systems and/or example methods described herein may detect the clusters related to internationalization configuration files of fifteen languages supported by the project. A developer that is working on internationalization support may be provided with a checklist of dependent files to ensure, for example, that the dependent files are modified correctly. An example benefit of the example systems and/or example methods described herein is that the analysis is language agnostic.

[0053] For yet another example in a third example project, a cluster of an external service API, internal service API, and data access layer code may be detected where the data access layer includes a plurality of configuration files that may be detected as part of a sub-graph of the cluster and another sub-graph of the cluster may contain changes in external and internal service APIs as well as the integration tests that are changed as a result.

[0054] For yet another example, a central component in a fourth example project may support two types of deployments (such as a local deployment and a remote deployment) where some of the code is different in order to handle these two deployment types, and thus, an extensible markup language (XML) configuration may, for example, need to be modified in response to a change in an annotation. In contrast to static code analysis or a runtime analysis that would likely not detect this type of dependency, the example systems and/or example methods described herein may, for example, detect the dependency between the XML configuration and the deployment annotation.

[0055] Although the flow diagrams of Figures 4-8 illustrate specific orders of execution, the order of execution may differ from that which is illustrated. For example, the order of execution of the blocks may be scrambled relative to the order shown. Also, the blocks shown in succession may be executed concurrently or with partial concurrence. All such variations are within the scope of the present description.

[0056] All of the features disclosed in this specification (including any accompanying claims, abstract and drawings), and/or all of the elements of any method or process so disclosed, may be combined in any combination, except combinations where at least some of such features and/or elements are mutually exclusive.

[0057] The present description has been shown and described with reference to the foregoing examples. It is understood, however, that other forms, details, and examples may be made without departing from the spirit and scope of the following claims. The use of the words "first," "second," or related terms in the claims are not used to limit the claim elements to an order or location, but are merely used to distinguish separate claim elements.

CLAIMS

What is claimed is:

1. A system comprising:
a dependency engine to:
identify a file pair of a first commit log entry of a plurality of commit log entries in a commit history, the first commit log entry containing a committer identifier and a first plurality of files that includes a first file and a second file;
assign a dependency rank to the file pair based on a number of times the first file and the second file appear together in the plurality of commit log entries and a number of distinct committers that commit the first file and the second file together;
a filter engine to compare the dependency rank of the file pair to a confidence rank; and
a cluster engine to indicate the second file is dependent on the first file when the dependency rank of the file pair achieves the confidence rank.

2. The system of claim 1, wherein:
the confidence rank is based on a project attribute of a project associated with the commit history, wherein the project attribute is at least one of a project size, a number of total commits of the project, a number of changes of a section of the project, and a number of developers of the project.

3. The system of claim 1, wherein:
the dependency engine is to:
create a dependency graph comprising a first node that represents the first file;
create a first edge between the first node and a second node that represents the second file; and

increase the dependency rank of the first edge when a second commit log entry of the plurality of commit log entries contains a second plurality of files comprising the first file and the second file;

the filter engine is to remove the first edge from the dependency graph when the dependency rank is below the confidence rank; and

the cluster engine is to identify a cluster of nodes that are connected in the dependency graph.

4. The system of claim 3, wherein:

the dependency engine determines the dependency rank as a weighted value based on a weight function using a set of parameters comprising a number of commits, a number of distinct committers that perform the commits, and a time decay function that emphasizes a first commit that is committed more recently than a second commit with a greater weighted value than the second commit.

5. The system of claim 3, wherein the cluster engine:

identifies a third node that lacks a direct edge to the first node, the cluster of nodes including the first node, the second node, and the third node; and

splits the cluster of nodes into a plurality of sub-graphs, wherein:

a first sub-graph of the plurality of sub-graphs comprises the first node, the second node, and the first edge connecting the first node and second node; and

a second sub-graph of the plurality of sub-graphs comprising the second node, the third node, and a second edge connecting the second node and the third node.

6. The system of claim 1, wherein the dependency engine is to:

identify a code pair when a first section of code changes in the first file and a second section of code changes in the second file;

determine a regularity of the code pair based on a number of times the first section of code and the second section of code change together in the commit history; and

modify the dependency rank associated with the file pair based on the regularity.

7. A non-transitory computer-readable storage medium comprising a set of instructions executable by a processor resource to cause the processor resource to:

modify a dependency rank of an edge of a dependency graph when a commit log entry contains a first file and a second file and the first file is represented by a first node of the dependency graph connected by the edge to a second node of the dependency graph representing the second file, the dependency rank being a value associated with a file pair represented by the edge;

compare the dependency rank to a confidence rank, the confidence rank based on a number of commit log entries in a commit history; and

indicate the first file and the second file have a dependency when the dependency rank achieves the confidence rank.

8. The medium of claim 6, wherein, when the commit log entry contains a plurality of files that includes the first file and the second file and the dependency graph does not contain the first node and the second node, the set of instructions is executable by the processor resource to:

add the first node to the dependency graph and the second node to the dependency graph; and

create the edge between the first node and the second node.

9. The medium of claim 6, wherein the set of instructions is executable by the processor resource to at least one of:

weight the dependency rank based on a number of distinct committers identified by a plurality of commit log entries that include the first file and the second file, wherein the commit log entry comprises a first data structure comprising a commit identifier, a committer identifier, and a list of files; and

weight the dependency rank based on a comparison of a time window to a commit time of the commit log entry, wherein the commit log entry comprises a

second data structure comprising the commit identifier, the committer identifier, the commit time, and the list of files.

10. The medium of claim 6, wherein the set of instructions is executable by the processor resource to:

identify a cluster of nodes in the dependency graph based on edges of nodes of the cluster; and

separate the cluster into a plurality of sub-graphs based on nodes of the cluster that do not have a direct connection, wherein shared nodes among a first sub-graph of the plurality of sub-graphs and a second sub-graph of the plurality of sub-graphs are duplicated.

11. A method for finding a dependency of files in a project comprising:

building a dependency graph having a plurality of nodes that represent a plurality of files in a commit history and a plurality of edges among the plurality of nodes based on whether the plurality of files represented by the plurality of nodes are contained in a commit log entry of the commit history;

filtering the dependency graph using a confidence rank to remove edges from the dependency graph that do not achieve the confidence rank;

creating a plurality of sub-graphs based on the filtered dependency graph; and

marking a first file of a first sub-graph of the plurality of sub-graphs as having dependency on a complementary set of files represented in the first sub-graph.

12. The method of claim 11, wherein building the dependency graph comprises:

creating a first node of the plurality of nodes that represents the first file when the first node does not exist in the dependency graph and the commit log entry contains the first file;

connecting, via a first edge of the plurality of edges of the dependency graph, the first node to a second node of the plurality of nodes that represents a second file when the first edge does not exist in the dependency graph and the commit log entry comprises the second file; and

associating the dependency rank with the first edge based on a number of distinct committers, a time decay function, and a number of commit log entries that include the first file and the second file.

13. The method of claim 11, wherein filtering the dependency graph comprises:

identifying at least one of a project size, a number of total commits of the project, and a number of committers of the project; and

determining the confidence rank based on the at least one of the project size, the number of total commits, and the number of committers.

14. The method of claim 11, wherein creating the plurality of sub-graphs comprises:

identifying a plurality of clusters of nodes, where a first cluster of a plurality of clusters is a first group of nodes that do not connect to a second group of nodes;

duplicating a first cluster when the first cluster contains a first node and a second node and there is not an edge between the first node and the second node; and

removing the first node from a first duplicate of the first cluster and the second node from a second duplicate of the first cluster.

15. The method of claim 11, comprising at least one of:

providing a list of dependency clusters in the project based on the plurality of sub-graphs;

providing metric information regarding the first sub-graph;

providing a list of dependency files of the first sub-graph associated with a checked-out file retrieved by a developer; and

providing a visual representation of the plurality of sub-graphs via a web page.

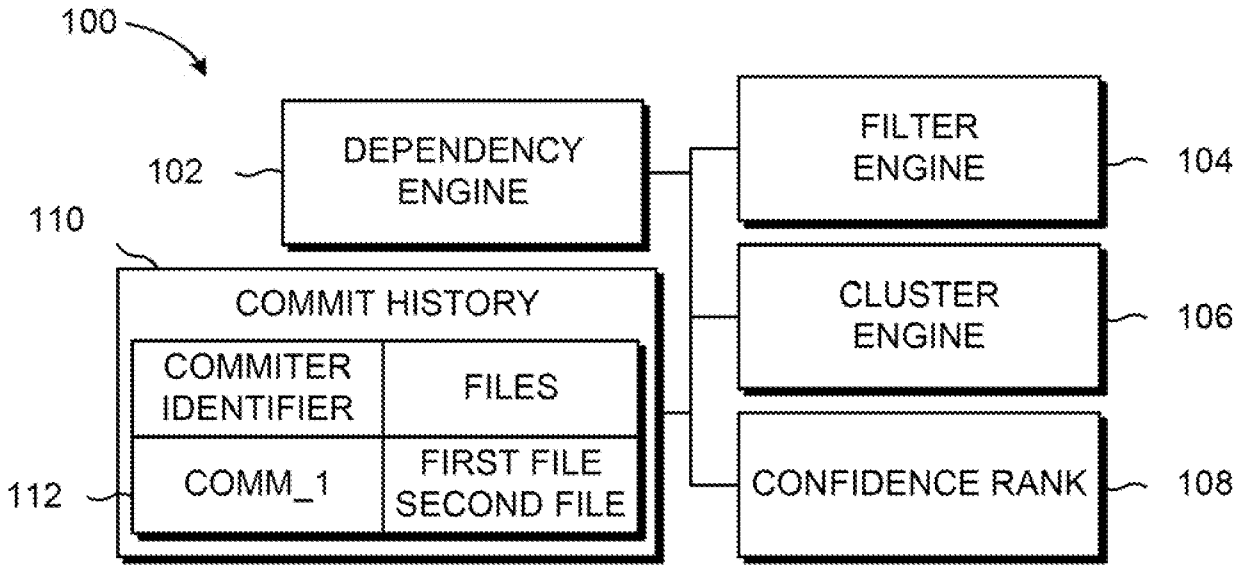
100
102
DEPENDENCY ENGINE
FILTER ENGINE
104
110
COMMIT HISTORY
COMMITER IDENTIFIER
FILES
CLUSTER ENGINE
106
112
COMM_1
FIRST FILE
SECOND FILE
CONFIDENCE RANK
108

FIG. 1

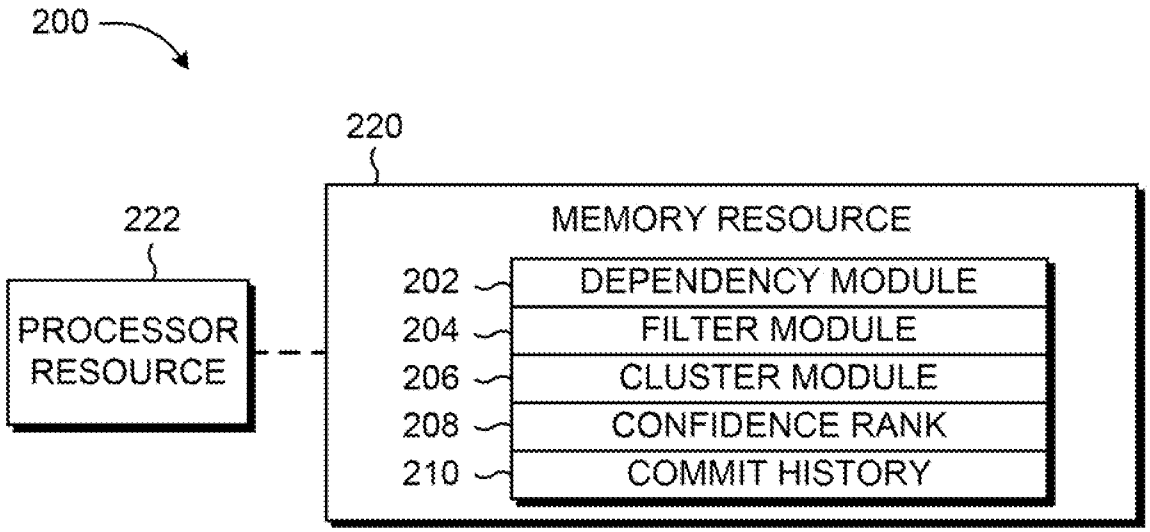
200
220
222
MEMORY RESOURCE
PROCESSOR RESOURCE
202
DEPENDENCY MODULE
204
FILTER MODULE
206
CLUSTER MODULE
208
CONFIDENCE RANK
210
COMMIT HISTORY

FIG. 2

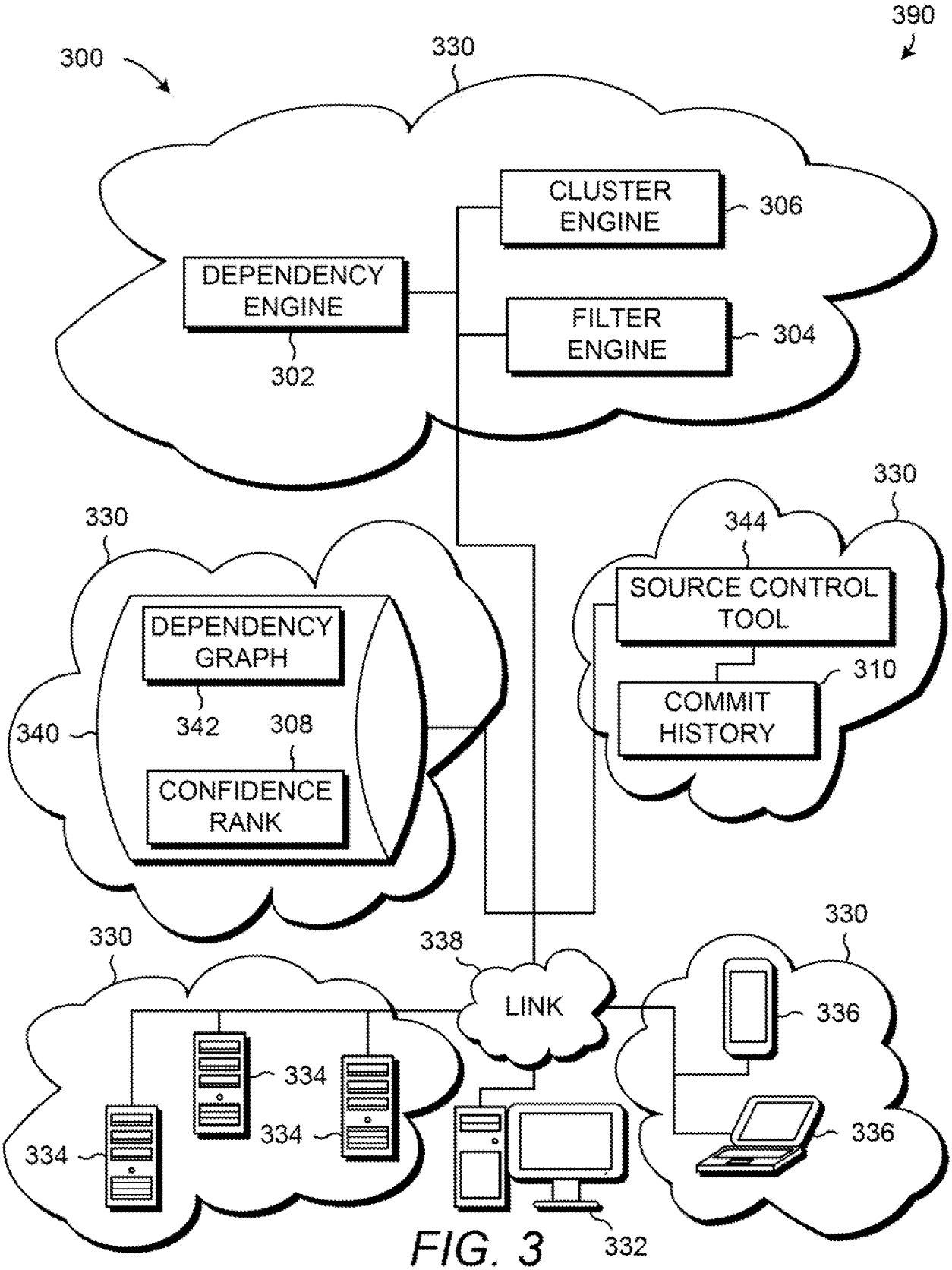
390
300
330
CLUSTER ENGINE
306
DEPENDENCY ENGINE
302
FILTER ENGINE
304
330
344
SOURCE CONTROL TOOL
310
COMMIT HISTORY
330
DEPENDENCY GRAPH
342
308
340
CONFIDENCE RANK
330
338
LINK
330
334
334
334
336
336
332

FIG. 3

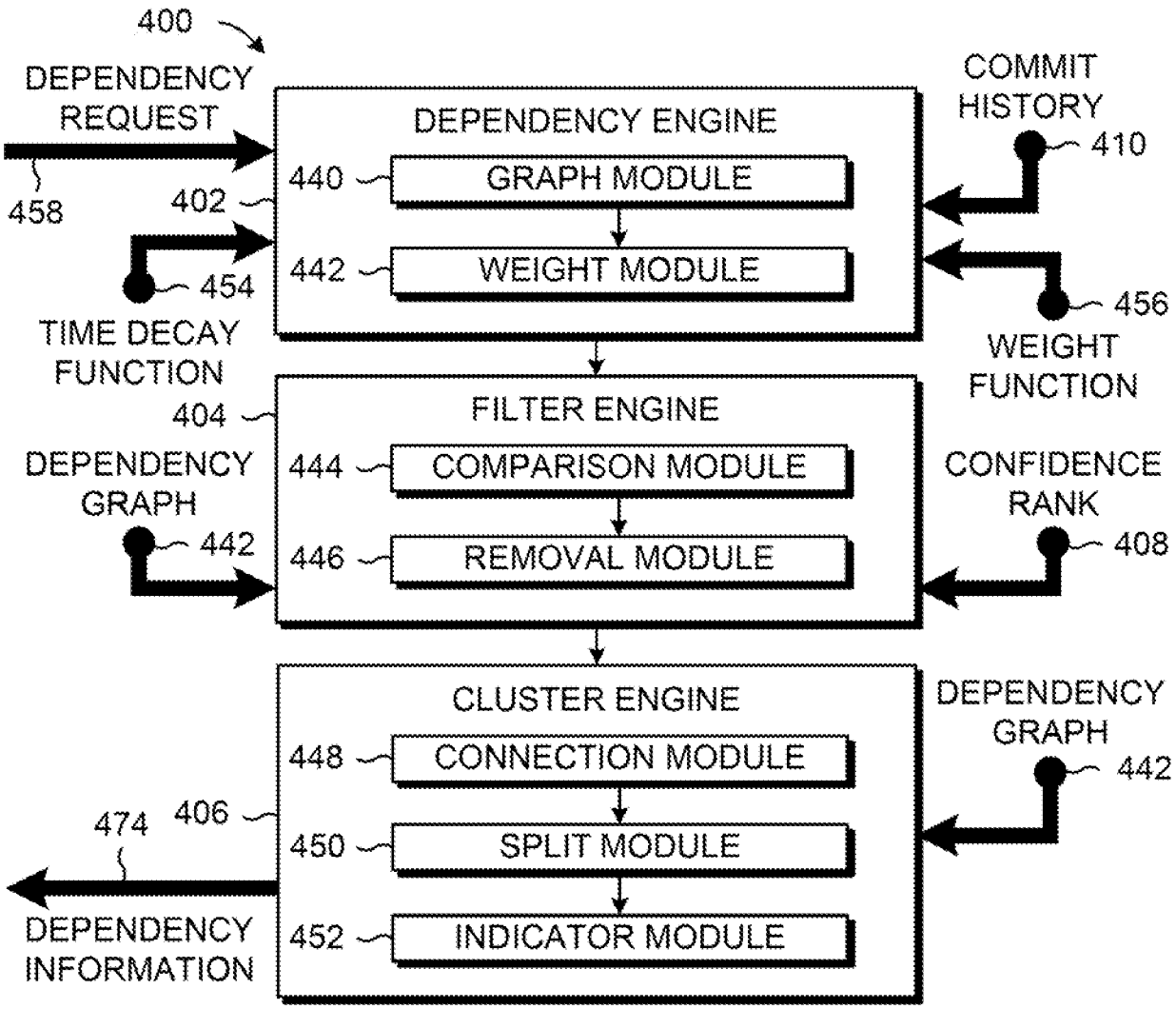
400
DEPENDENCY REQUEST
458
402
454
TIME DECAY FUNCTION
DEPENDENCY ENGINE
440
GRAPH MODULE
442
WEIGHT MODULE
COMMIT HISTORY
410
456
WEIGHT FUNCTION
404
FILTER ENGINE
DEPENDENCY GRAPH
442
444
COMPARISON MODULE
446
REMOVAL MODULE
CONFIDENCE RANK
408
CLUSTER ENGINE
448
CONNECTION MODULE
450
SPLIT MODULE
452
INDICATOR MODULE
DEPENDENCY GRAPH
442
406
474
DEPENDENCY INFORMATION

FIG. 4

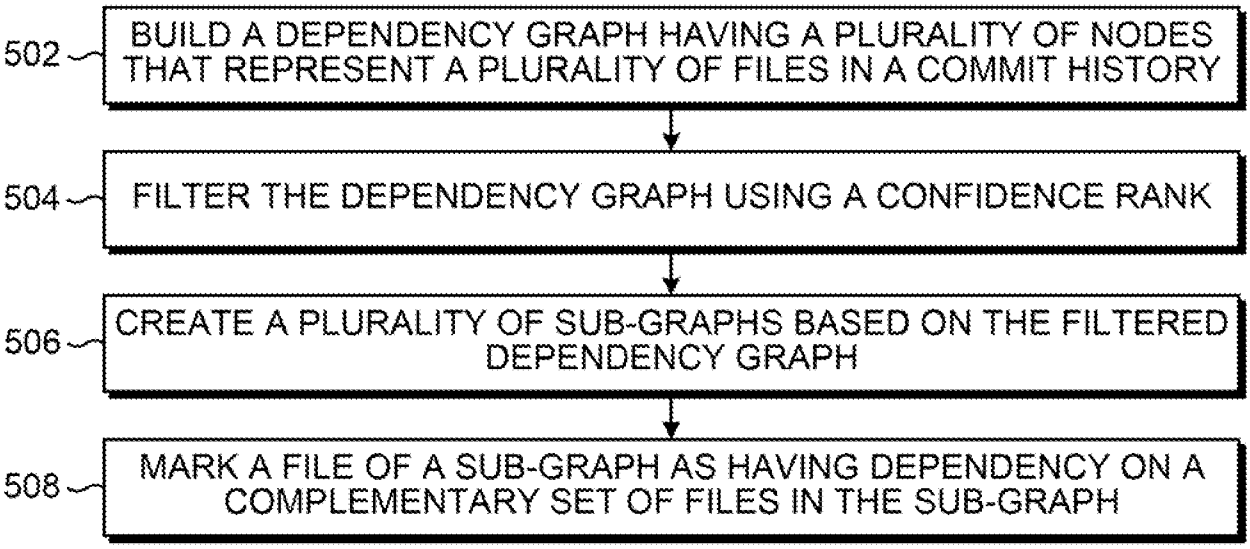
502
BUILD A DEPENDENCY GRAPH HAVING A PLURALITY OF NODES THAT REPRESENT A PLURALITY OF FILES IN A COMMIT HISTORY
504
FILTER THE DEPENDENCY GRAPH USING A CONFIDENCE RANK
506
CREATE A PLURALITY OF SUB-GRAPHS BASED ON THE FILTERED DEPENDENCY GRAPH
508
MARK A FILE OF A SUB-GRAPH AS HAVING DEPENDENCY ON A COMPLEMENTARY SET OF FILES IN THE SUB-GRAPH

FIG. 5

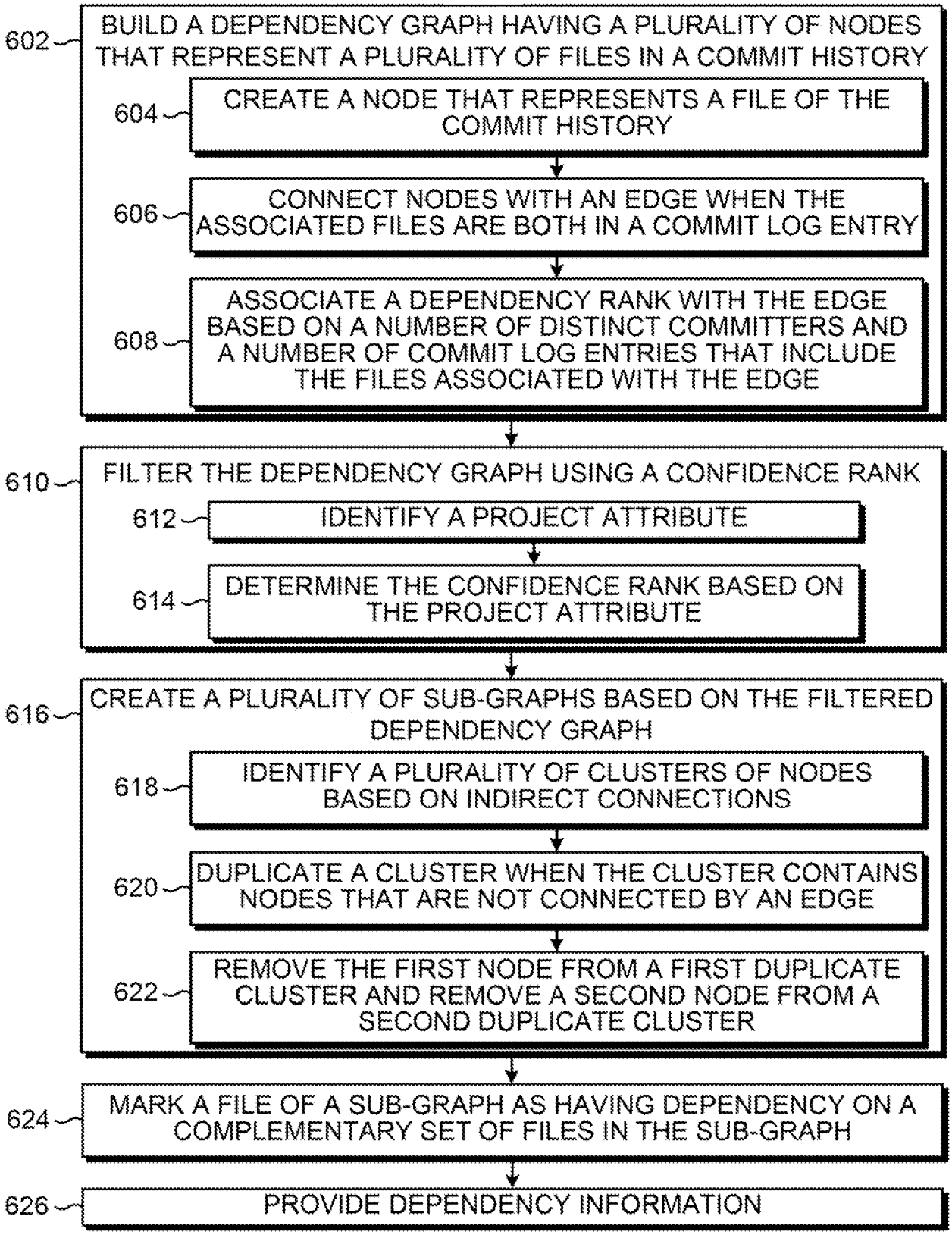
602
BUILD A DEPENDENCY GRAPH HAVING A PLURALITY OF NODES THAT REPRESENT A PLURALITY OF FILES IN A COMMIT HISTORY
604
CREATE A NODE THAT REPRESENTS A FILE OF THE COMMIT HISTORY
606
CONNECT NODES WITH AN EDGE WHEN THE ASSOCIATED FILES ARE BOTH IN A COMMIT LOG ENTRY
608
ASSOCIATE A DEPENDENCY RANK WITH THE EDGE BASED ON A NUMBER OF DISTINCT COMMITTERS AND A NUMBER OF COMMIT LOG ENTRIES THAT INCLUDE THE FILES ASSOCIATED WITH THE EDGE
610
FILTER THE DEPENDENCY GRAPH USING A CONFIDENCE RANK
612
IDENTIFY A PROJECT ATTRIBUTE
614
DETERMINE THE CONFIDENCE RANK BASED ON THE PROJECT ATTRIBUTE
616
CREATE A PLURALITY OF SUB-GRAPHS BASED ON THE FILTERED DEPENDENCY GRAPH
618
IDENTIFY A PLURALITY OF CLUSTERS OF NODES BASED ON INDIRECT CONNECTIONS
620
DUPLICATE A CLUSTER WHEN THE CLUSTER CONTAINS NODES THAT ARE NOT CONNECTED BY AN EDGE
622
REMOVE THE FIRST NODE FROM A FIRST DUPLICATE CLUSTER AND REMOVE A SECOND NODE FROM A SECOND DUPLICATE CLUSTER
624
MARK A FILE OF A SUB-GRAPH AS HAVING DEPENDENCY ON A COMPLEMENTARY SET OF FILES IN THE SUB-GRAPH
626
PROVIDE DEPENDENCY INFORMATION

FIG. 6

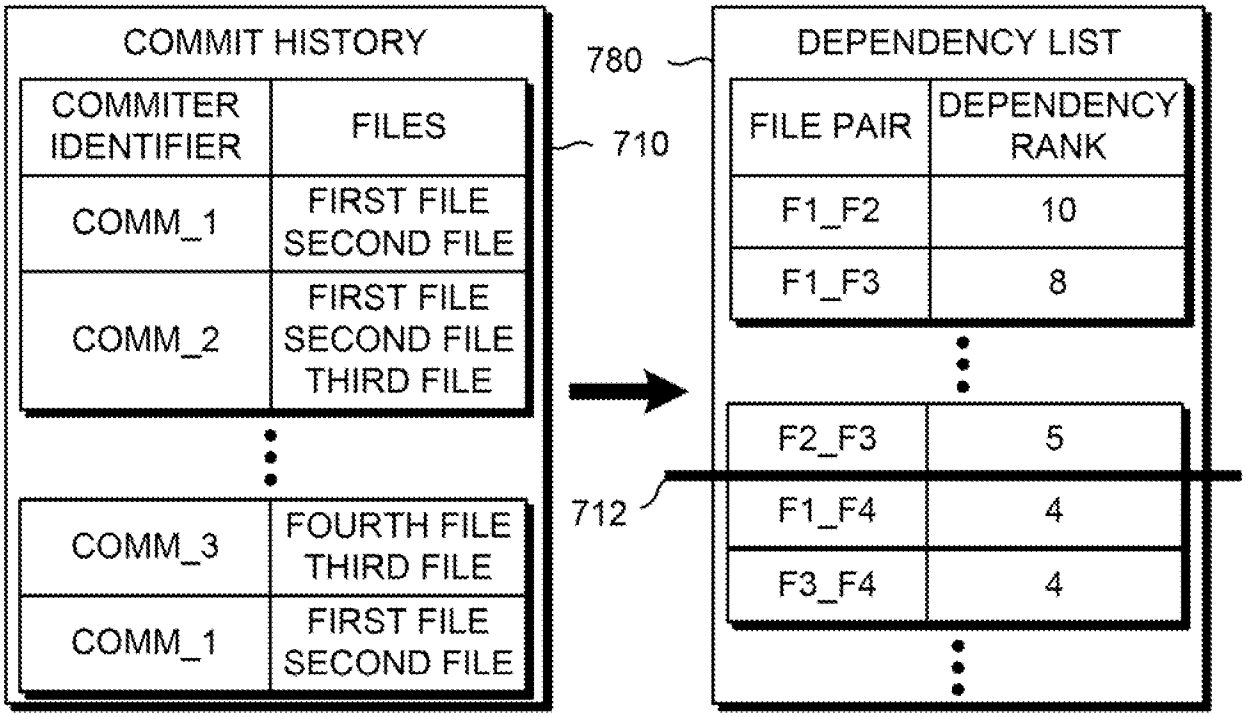
COMMIT HISTORY
COMMITER IDENTIFIER
FILES
COMM_1
FIRST FILE
SECOND FILE
COMM_2
FIRST FILE
SECOND FILE
THIRD FILE
COMM_3
FOURTH FILE
THIRD FILE
COMM_1
FIRST FILE
SECOND FILE
780
710
712
DEPENDENCY LIST
FILE PAIR
DEPENDENCY RANK
F1_F2
10
F1_F3
8
F2_F3
5
F1_F4
4
F3_F4
4

FIG. 7

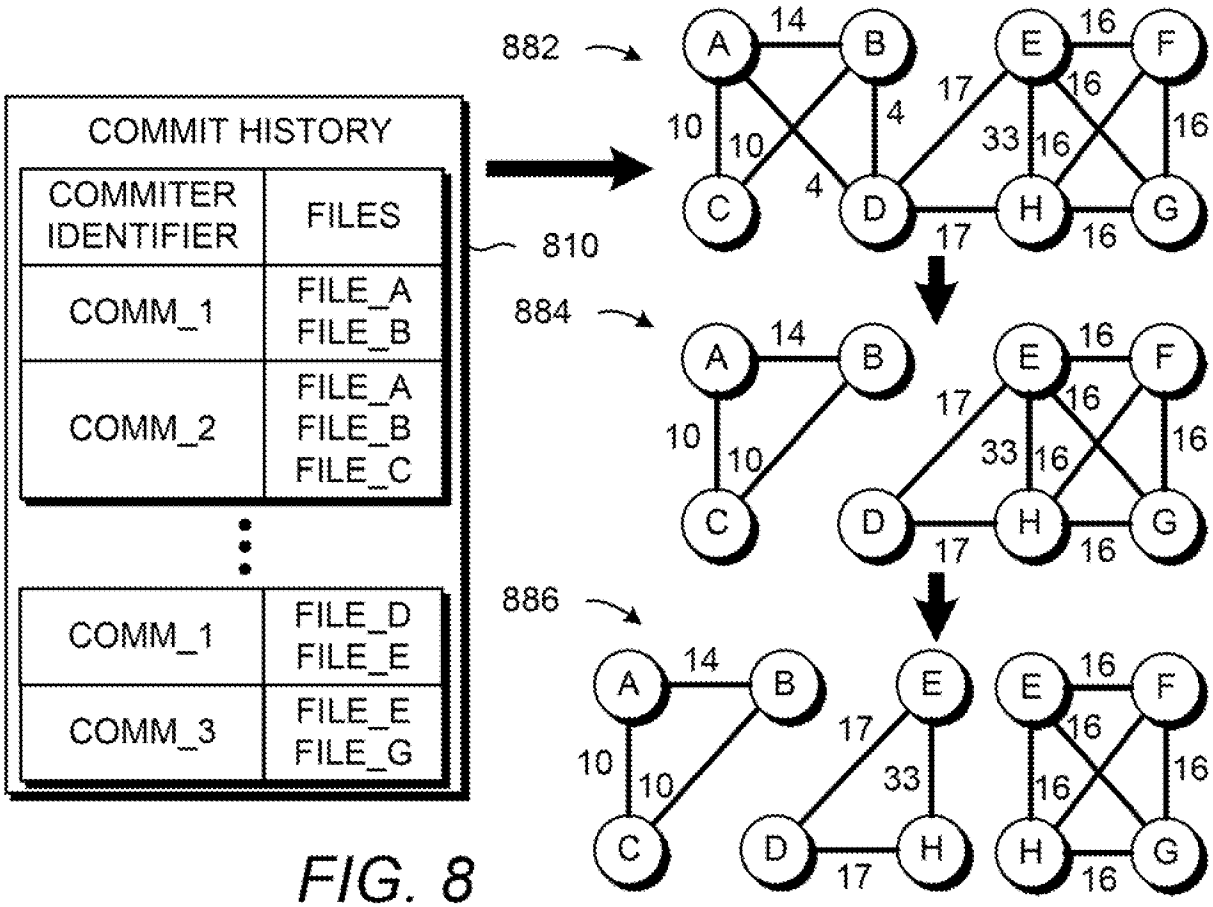
COMMIT HISTORY
COMMITER IDENTIFIER
FILES
COMM_1
FILE_A
FILE_B
COMM_2
FILE_A
FILE_B
FILE_C
COMM_1
FILE_D
FILE_E
COMM_3
FILE_E
FILE_G
810
882
884
886
A
B
C
D
E
F
G
H
14
10
10
4
4
17
33
16
16
16
16
16
17
17

FIG. 8

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2015/032923

A. CLASSIFICATION OF SUBJECT MATTER

G06F 17/30(2006.01)i, G06F 17/00(2006.01)i

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F 17/30; G06F 9/44; G06F 9/46; G06F 1/24; G06F 17/00

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords:software development, file, source code, commit log entry, commit history, dependency rank, confidence rank, dependency graph, sub-graph, node, edge, nodes cluster

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	DIANE J. HU et al. `Latent Variable Models for Predicting File Dependencies in Large-Scale Software Development.` University of California, San Diego. 14 June 2011. Retrieved from: http://cseweb.ucsd.edu/~lerner/papers/nips10.pdf See abstract; page 1, line 28 - page 8, line 34.	1-10
A		11-15
Y	US 2008-0104570 A1 (CHRISTOPHER CHEDGEY et al.) 01 May 2008 See paragraphs [0026], [0060], [0067], [0111], [0118]-[0121], and [0135]; and figures 6, 11, and 13.	1-10
Y	US 2014-0109106 A1 (MICHAEL C. FANNING et al.) 17 April 2014 See paragraphs [0003]-[0005], [0022], [0035], and [0056].	6,8
A	US 2012-0296878 A1 (MASAYUKI NAKAE et al.) 22 November 2012 See paragraphs [0002], [0057], [0094], and [0097].	1-15
A	US 2004-0177244 A1 (RICHARD C. MURPHY et al.) 09 September 2004 See paragraphs [0010], [0036], [0084], and [0115].	1-15

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
30 March 2016 (30.03.2016)	**31 March 2016 (31.03.2016)**

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 35208, Republic of Korea
Facsimile No. +82-42-481-8578

Authorized officer
HAN, JOONG SUB
Telephone No. +82-42-481-3578

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2015/032923

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2008-0104570 A1	01/05/2008	AU 2001-22152 A1	03/07/2001
		AU 2215201 A	03/07/2001
		IE S20010131 A2	30/05/2001
		US 2004-0205726 A1	14/10/2004
		US 7409679 B2	05/08/2008
		WO 01-046798 A2	28/06/2001
		WO 01-046798 A3	12/09/2002
		WO 01-046798 A8	22/04/2004
US 2014-0109106 A1	17/04/2014	US 9122490 B2	01/09/2015
		WO 2014-062950 A1	24/04/2014
US 2012-0296878 A1	22/11/2012	JP 05644777 B2	24/12/2014
		WO 2011-089864 A1	28/07/2011
US 2004-0177244 A1	09/09/2004	US 7152157 B2	19/12/2006
		WO 2004-079534 A2	16/09/2004
		WO 2004-079534 A3	22/12/2005