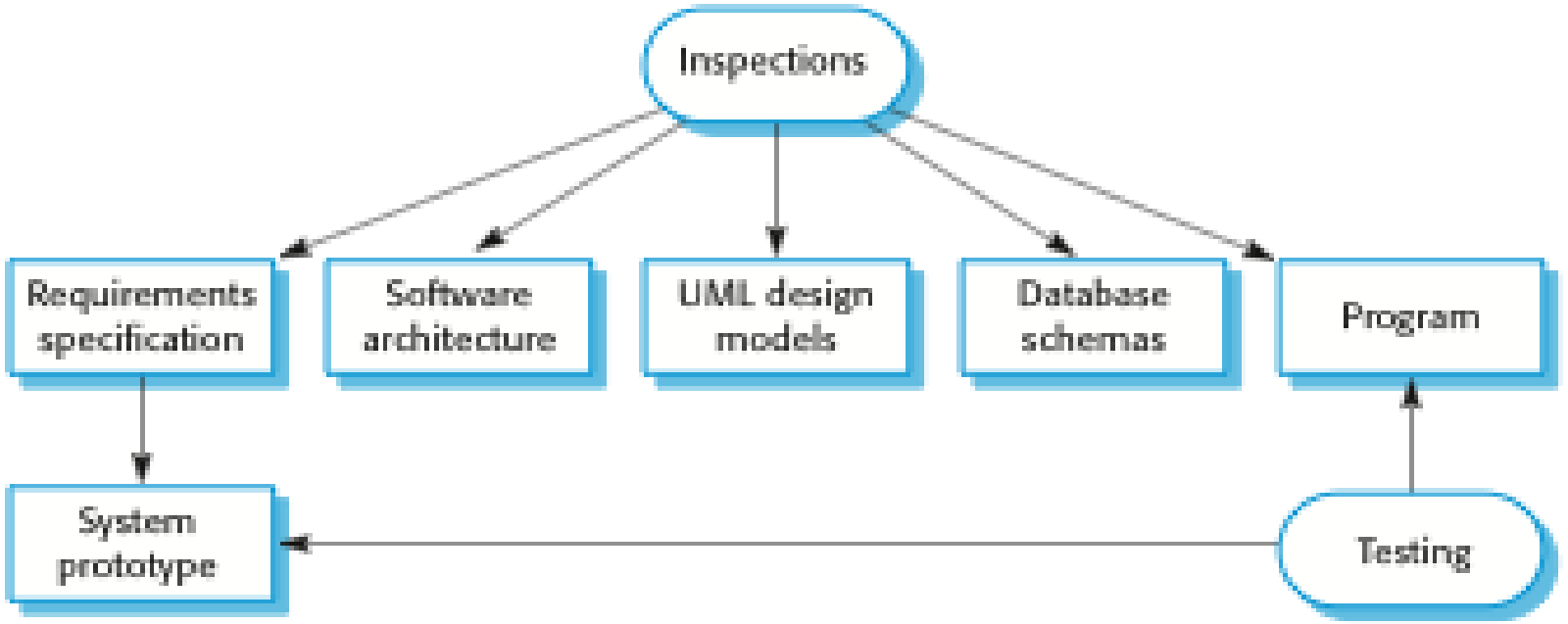
Verification vs validation

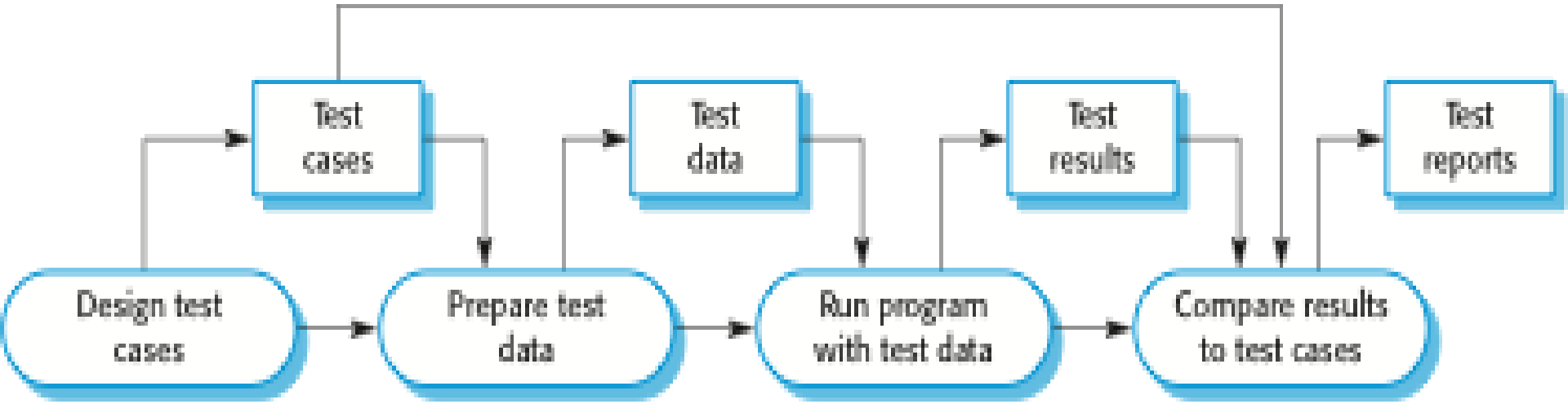
- **Verification**:
  "Are we building the product right".
- The software should conform to its specification.
- **Validation**:
  "Are we building the right product".
- The software should do what the user really requires.

Inspections and testing

- **Software inspections** Concerned with **analysis of the static system representation** to discover problems **(static verification)**
- **Software testing** Concerned with exercising and **observing product behaviour (dynamic verification)**
  - The system is executed with test data and its operational behaviour is observed.



A model of the software testing process



Software inspections

- These involve people **examining the source representation** with the aim of discovering anomalies and defects.
- Inspections **not require execution of a system so may be used before implementation.**
- They may be **applied to any representation of the system** (requirements, design, configuration data, test data, etc.).
- They have been shown to be an **effective technique for discovering program errors.**
- **Inspections can check conformance with a specification** but **not conformance with the customer's real requirements**.
- Inspections **cannot check non-functional characteristics such as performance, usability,** etc.

Advantages of inspections

- **During testing, errors can mask (hide) other errors**. Because **inspection** is a static process, you **don't have to be concerned with interactions between errors**.
- **Incomplete versions of a system can be inspected without additional costs**. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an **inspection can also consider** broader **quality attributes** of a program, such as **compliance with standards, portability and maintainability.**

Stages of testing

- **Development testing**, where the system is **tested during development** to discover bugs and defects.
- **Release testing**, where **a separate testing team test a complete version of the system before it is released** to users.
- **User testing**, where users or potential **users of a system test the system in their own environment.**

Development testing

- Development testing includes **all testing activities that are carried out by the team developing the system**.
  - **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  - **System testing**, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Unit testing

- Unit testing is the process of **testing individual components in isolation.**
- It is **a defect testing process**.
- **Units** may be:
  - **Individual functions or methods within an object**
  - **Object classes with several attributes and methods**
  - Composite components with defined interfaces used to access their functionality.

Object class testing

- **Complete test coverage of a class involves**
  - **Testing all operations** associated with an object
  - Setting and **interrogating all object attributes**
  - **Exercising the object in all possible states**.
- **Inheritance makes it more difficult to design object class tests** as the information to be tested is not localised.

Component Testing

- **Testing composite components should therefore focus on showing that the component interface behaves according to its specification.**

Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- **Interface types**
  - **Parameter interfaces** Data passed from one method or procedure to another.
  - **Shared memory interfaces** Block of memory is shared between procedures or functions.
  - **Procedural interfaces** Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - **Message passing interfaces** Sub-systems request services from other sub-systems

Interface errors

- **Interface misuse**
  - **A calling component calls another component and makes an error in its use of its interface** e.g. parameters in the wrong order.
- **Interface misunderstanding**
  - **A calling component embeds assumptions about the behaviour of the called component which are incorrect.**
- **Timing errors**
  - **The called and the calling component operate at different speeds** and out-of-date information is accessed.

Interface testing guidelines

- **Design tests so that parameters to a called procedure are at the extreme ends of their ranges.**
- Always **test pointer parameters with null pointers**.
- **Design tests which cause the component to fail**.
- **Use stress testing in message passing systems**.
- In shared memory systems, vary the order in which components are activated.

System testing

- System testing **during development** involves **integrating components to create a version of the system** and then **testing the integrated system.**
- The focus in system testing is **testing the interactions between components**.
- System testing **checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.**
- System testing **tests the emergent behavior of a system.**

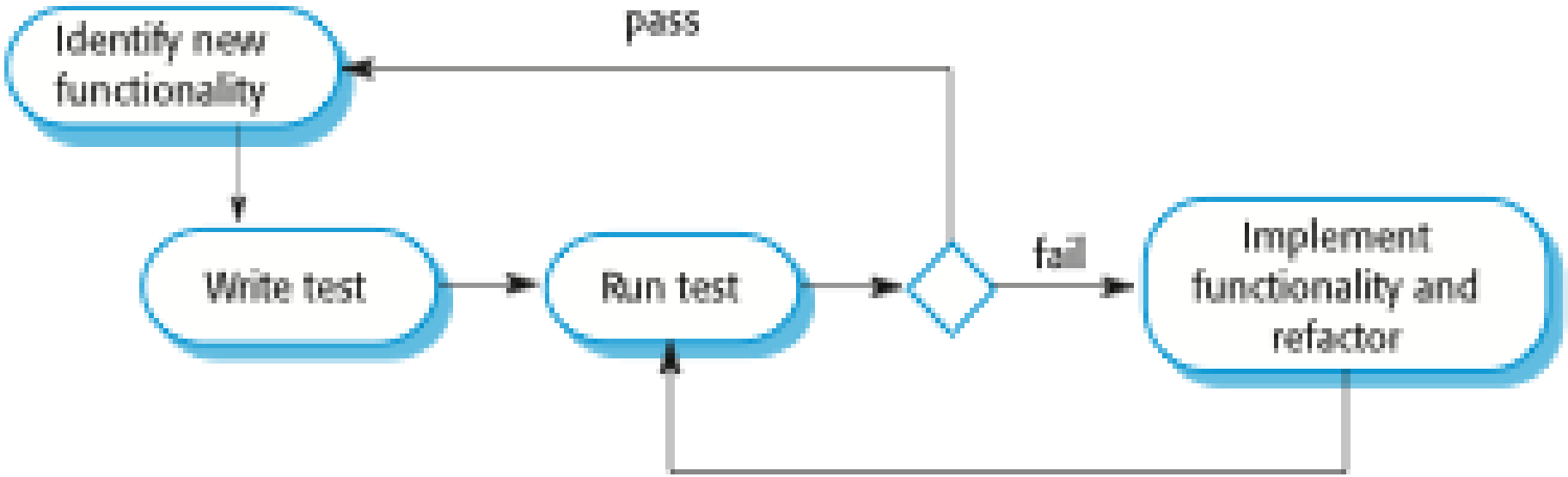
Use-case testing

- **The use-cases developed to identify system interactions can be used as a basis for system testing.**
- **Each use case usually involves several system components** so testing the use case forces these interactions to occur.
- **The sequence diagrams associated with the use case documents the components and interactions** that are being tested.

Test-driven development

- Test-driven development **(TDD) is an approach to program development in which you inter-leave testing and code development**.
- **Tests are written before code** and ‘passing’ the tests is the critical driver of development.
- **You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test.**
- TDD was introduced as **part of agile methods such as Extreme Programming**. However, it can also be used in plan-driven development processes.

Test-driven development



Benefits of test-driven development

- **Code coverage**
  - **Every code segment** that you write **has at least one associated test** so all code written has at least one test.
- **Regression testing**
  - A regression test suite is **developed incrementally as a program is developed.**
- **Simplified debugging**
  - **When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.**
- **System documentation**
  - **The tests themselves are a form of documentation** that describe what the code should be doing.

Regression testing

- Regression testing is **testing the system to check that changes have not 'broken' previously working code**.
- In a manual testing process, regression testing is expensive but, **with automated testing**, it is simple and straightforward. **All tests are rerun every time a change is made to the program.**
- **Tests must run 'successfully' before the change is committed.**

Release testing

- **Release testing** is the process of **testing a particular release of a system** that is **intended for use outside of the development team.**
- The **primary goal** of the release testing process is **to convince the supplier of the system that it is good enough for use.**
  - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- **Release testing is usually a black-box testing process where tests are only derived from the system specification.**

Release testing and system testing

- **Release testing** is a **form of system testing**.
- **Important differences**:
  - **A separate team that has not been involved in the system development, should be responsible for release testing.**
  - **System testing by the development team should focus on discovering bugs in the system (defect testing).**
  - The **objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).**

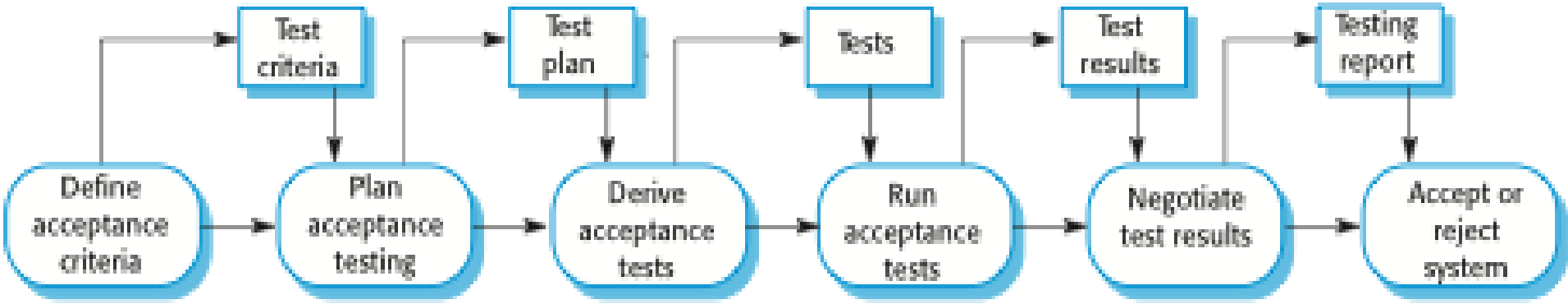
User testing

- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

Types of user testing

- **Alpha testing**
  - **Users** of the software work **with the development team** to **test the software at the developer's site.**
- **Beta testing**
  - A **release of the software is made available to users to allow them to experiment and to raise problems** that they discover with the system developers.
- **Acceptance testing**
  - **Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.** Primarily for custom systems.

The acceptance testing process



Agile methods and acceptance testing

- In **agile** methods, the **user/customer is part of the development team** and is responsible for making decisions on the acceptability of the system.
- **Tests are defined by the user/customer** and are integrated with other tests in that **they are run automatically when changes are made**.
- **There is no separate acceptance testing process**.
- **Main problem** here is **whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.**

Software quality management

- Concerned with **ensuring that the required level of quality is achieved in a software product**.
- **Three principal concerns**:
  - **At the organizational level**, quality management is concerned with **establishing a framework of organizational processes and standards** that will lead to high-quality software.
  - **At the project level**, quality management involves
    - **Establishing a quality plan for a project**. The quality **plan should set out the quality goals** for the project **and define what processes and standards are to be used**.
    - **Application of specific quality processes and checking that these planned processes have been followed**.

Quality management activities

- Quality management provides **an independent check on the software development process**.
- The quality management process **checks the project deliverables to ensure that they are consistent with organizational standards and goals**
- **The quality team should be independent from the development team** so that they can take an objective view of the software.

  This allows them to report on software quality without being influenced by software development issues.

Quality management and software development

Quality planning

- **A quality plan sets out the desired product qualities and how these are assessed and defines the most significant quality attributes**.
- The quality plan should **define the quality assessment process.**
- It should **set out which organisational standards should be applied** and, where necessary, define new standards to be used.

Software quality

- **Quality**, simplistically, **means that a product should meet its specification.**
- **This is problematical for software systems**
  - **There is a tension between customer quality requirements (efficiency, reliability**, etc.) **and developer quality requirements (maintainability, reusability**, etc.);
  - **Some quality requirements are difficult to specify in an unambiguous way**;
  - **Software specifications are usually incomplete and often inconsistent.**
- The **focus may be 'fitness for purpose' rather than specification conformance.**

Software fitness for purpose

- **Have programming and documentation standards been followed** in the development process?
- **Has the software been properly tested**?
- **Is the software sufficiently dependable** to be put into use?
- **Is the performance of the software acceptable** for normal use?
- **Is the software usable**?
- **Is the software well-structured and understandable**?

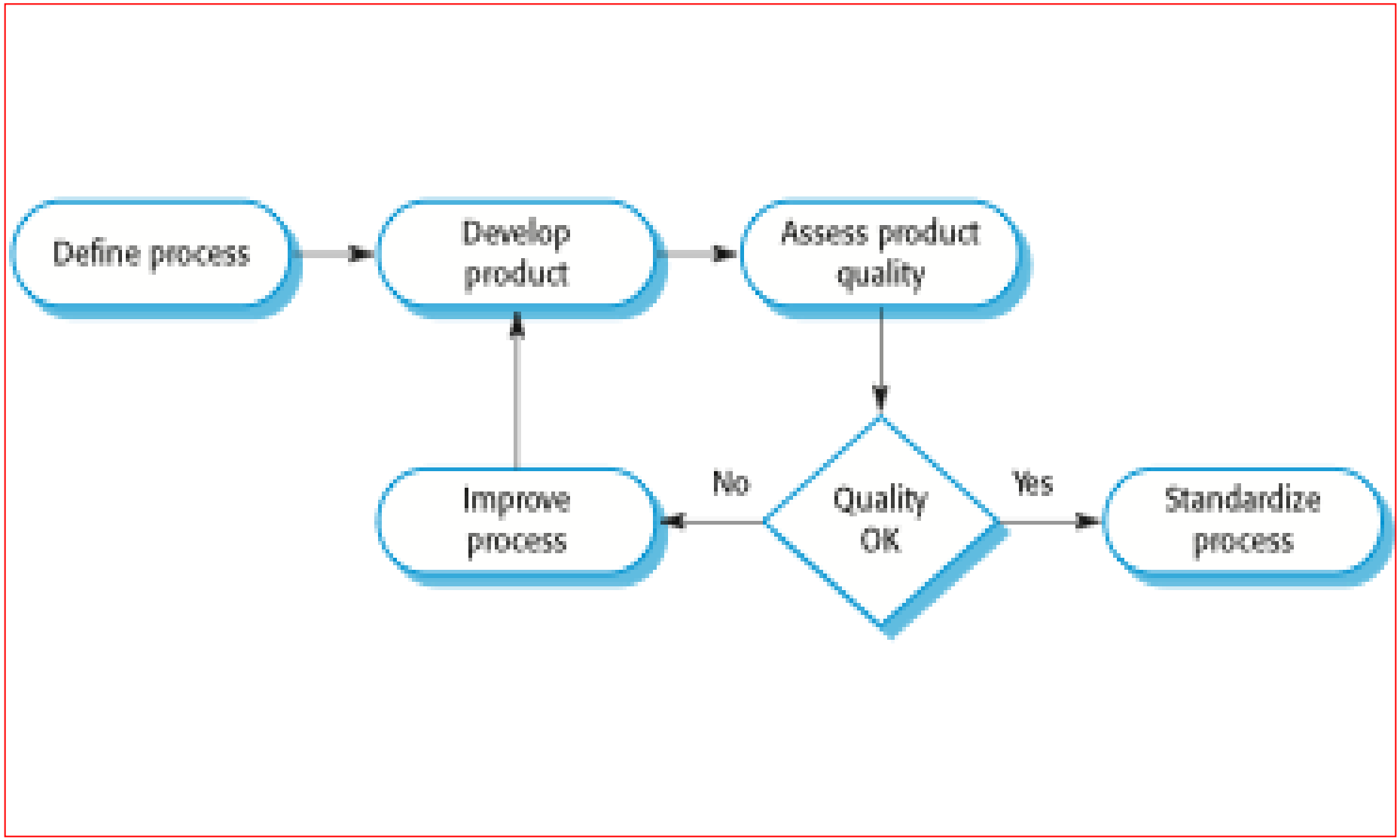
Quality conflicts

- **It is not possible for any system to be optimized for all of software quality attributes** – for example, **improving robustness may lead to loss of performance**.
- The **quality plan should therefore define the most important quality attributes** for the software that is being developed.
- **The plan should also include a definition of the quality assessment process**, an agreed way of **assessing whether** some quality, such as **maintainability or robustness, is present in the product.**

Process and product quality

- The **quality of a developed product is influenced by the quality of the production process**.
- This is important in software development **as some product quality attributes are hard to assess**.
- However, there is a very **complex** and poorly understood **relationship between software processes and product quality**.
  - The **application of individual skills and experience** is particularly **important in software development**;
  - **External factors** such as the **novelty of an application or** the need for an **accelerated development** schedule **may impair product quality**.

Process-based quality



Software standards

- **Standards define the required attributes of a product or process**. They play an important role in quality management.
- Standards may be **international**, **national**, **organizational** or **project standards**.
- **Product standards define characteristics that all software components should exhibit**

  e.g. **a common programming style**.
- **Process standards define how the software process should be enacted**.

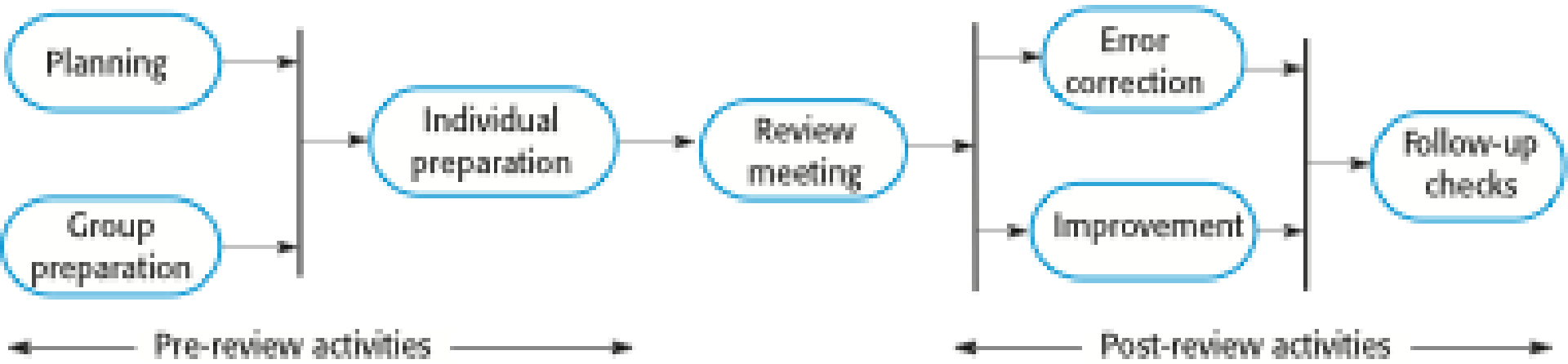
Reviews and inspections

- **A group examines part or all of a process or system and its documentation** to find potential problems.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- There are **different types of review** with different objectives
  - **Inspections for defect removal (product);**
  - **Reviews for progress assessment (product and process);**
  - **Quality reviews (product and standards)**.

Quality reviews

- **A group of people carefully examine part or all of a software system and its associated documentation.**
- **Code, designs, specifications, test plans, standards, etc. can all be reviewed.**
- **Software or documents may be 'signed off' at a review** which signifies that **progress to the next development stage has been approved** by management.

The software review process



Reviews and agile methods

- The **review process in agile software development** is **usually informal**.
  - **In Scrum**, for example, **there is a review meeting after each iteration of the software has been completed (a sprint review)**, where quality issues and problems may be discussed.
- **In extreme programming, pair programming ensures that code is constantly being examined and reviewed by another team member.**
- **XP relies on individuals taking the initiative to improve and refactor code**. **Agile approaches are not usually standards-driven**, so issues of standards compliance are not usually considered.

Program inspections

- These **are peer reviews where engineers examine the source of a system with the aim of discovering anomalies** and defects.
- **Inspections do not require execution of a system** so may be **used before implementation**.
- They **may be applied to any representation of the system** (**requirements, design, configuration data, test data**, etc.).
- They have been shown to be an **effective technique for discovering program errors**.

Inspection checklists

- **Checklist of common errors** should be **used to drive the inspection.**
- **Error checklists** are **programming language dependent** and **reflect the characteristic errors** that are likely to arise **in the language**.
- In general, the 'weaker' the type checking, the larger the checklist.
- **Examples: Initialisation, Constant naming, loop termination, array bounds**, etc.

Agile methods and inspections

- **Agile processes rarely use formal inspection or peer review processes**.
- **Rather, they rely on team members cooperating to check each other's code**, and informal guidelines, such as 'check before check-in', which suggest that programmers should check their own code.
- **Extreme programming** practitioners **argue that pair programming is an effective substitute for inspection** as this is, in effect, a continual inspection process.
- **Two people look at every line of code and check it before it is accepted.**

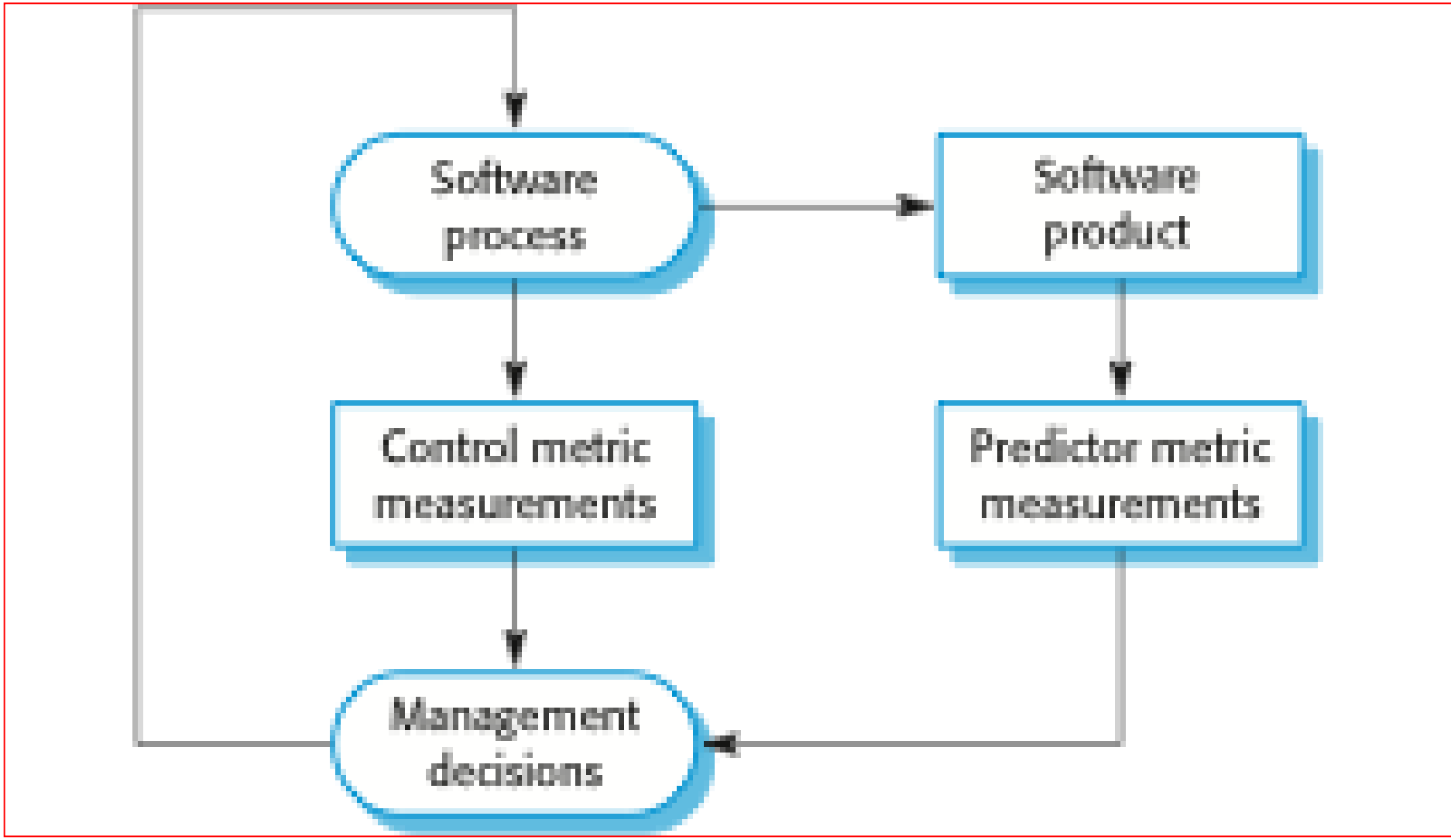
Software measurement and metrics

- **Software measurement** is concerned with **deriving a numeric value for an attribute of a software product or process**.

Software metric

- **Any type of measurement which relates to a software system**, process or related documentation
  - **Lines of code** in a program, the **Fog index**, number of **person-days** required to develop a component.
- Allow the software and the software process to be quantified.
- **May be used to predict product attributes or to control the software process**.

Predictor and control measurements



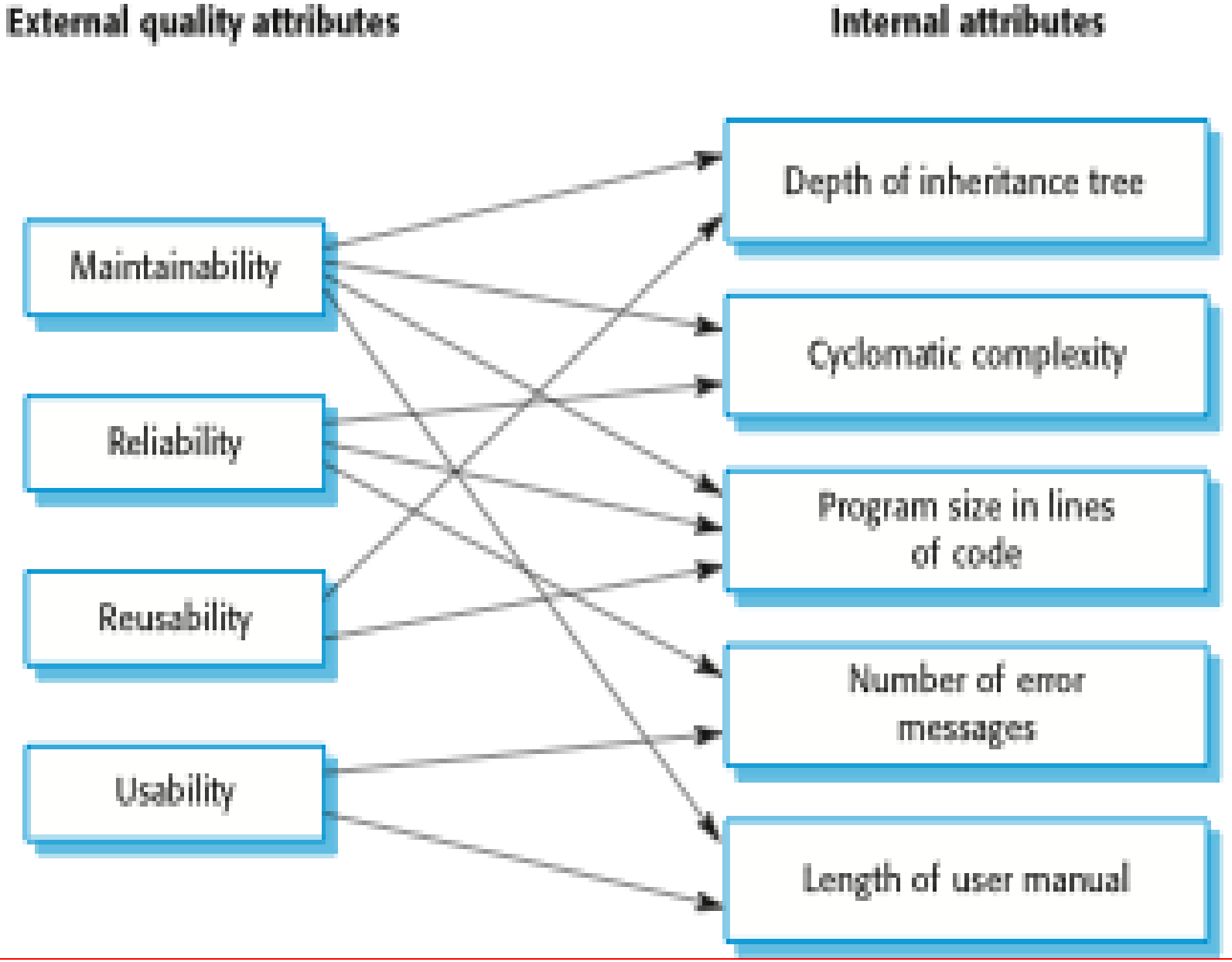
Use of measurements

- **To assign a value to system quality attributes**
  - **By measuring the characteristics of system components**, such as their **cyclomatic complexity**, **you can assess system quality attributes**, **such as maintainability.**
- **To identify the system components whose quality is sub-standard**
  - you can measure **components** to discover those **with the highest complexity**. These are **most likely to contain bugs**

Metrics assumptions

- **The relationship exists between what we can measure and what we want to know**. **We can only measure internal attributes**

Relationships between internal and external software



Product metrics

- A **quality metric** should be a **predictor of product quality.**
- **Classes of product metric**
  - **Dynamic metrics** which are **collected by measurements made of a program in execution**;
  - **Static metrics** which are **collected by measurements made of the system representations**;

Dynamic metrics help **assess efficiency and reliability**

Static metrics help **assess complexity, understandability and maintainability.**

Dynamic and static metrics

- **Dynamic metrics** are closely **related to software quality attributes**
  - It is relatively **easy to measure** the **response time of a system (performance attribute)** or the **number of failures (reliability attribute).**
- **Static metrics have an indirect relationship with quality attributes**
  - You need to try and **derive a relationship between these metrics and properties such as complexity, understandability and maintainability.**

Static software product metrics

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.

Static software product metrics

Software metric	Description
Cyclomatic complexity	This is **a measure of the control complexity of a program.** This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is **a measure of the average length of identifiers (names for variables, classes, methods, etc.)** in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is **a measure of the depth of nesting of if-statements in a program**. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is **a measure of the average length of words and sentences in documents**. The higher the value of a document's Fog index, the more difficult the document is to understand.

The CK object-oriented metrics suite

Object-oriented metric	Description
Weighted methods per class (WMC)	This is **the number of methods in each class**, **weighted by the complexity of each method**. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. **The larger the value for this metric, the more complex the object class**. Complex objects are more likely to be **difficult to understand.** They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents **the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design**. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is **a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse**. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

The CK object-oriented metrics suite

Object-oriented metric	Description
Coupling between object classes (CBO)	**Classes are coupled when methods in one class use methods or instance variables defined in a different class**. **CBO is a measure of how much coupling exists**. **A high value for CBO means that classes are highly dependent**, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	**RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class**. Again, RFC is related to complexity. **The higher the value for RFC, the more complex a class** and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. **LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes**. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

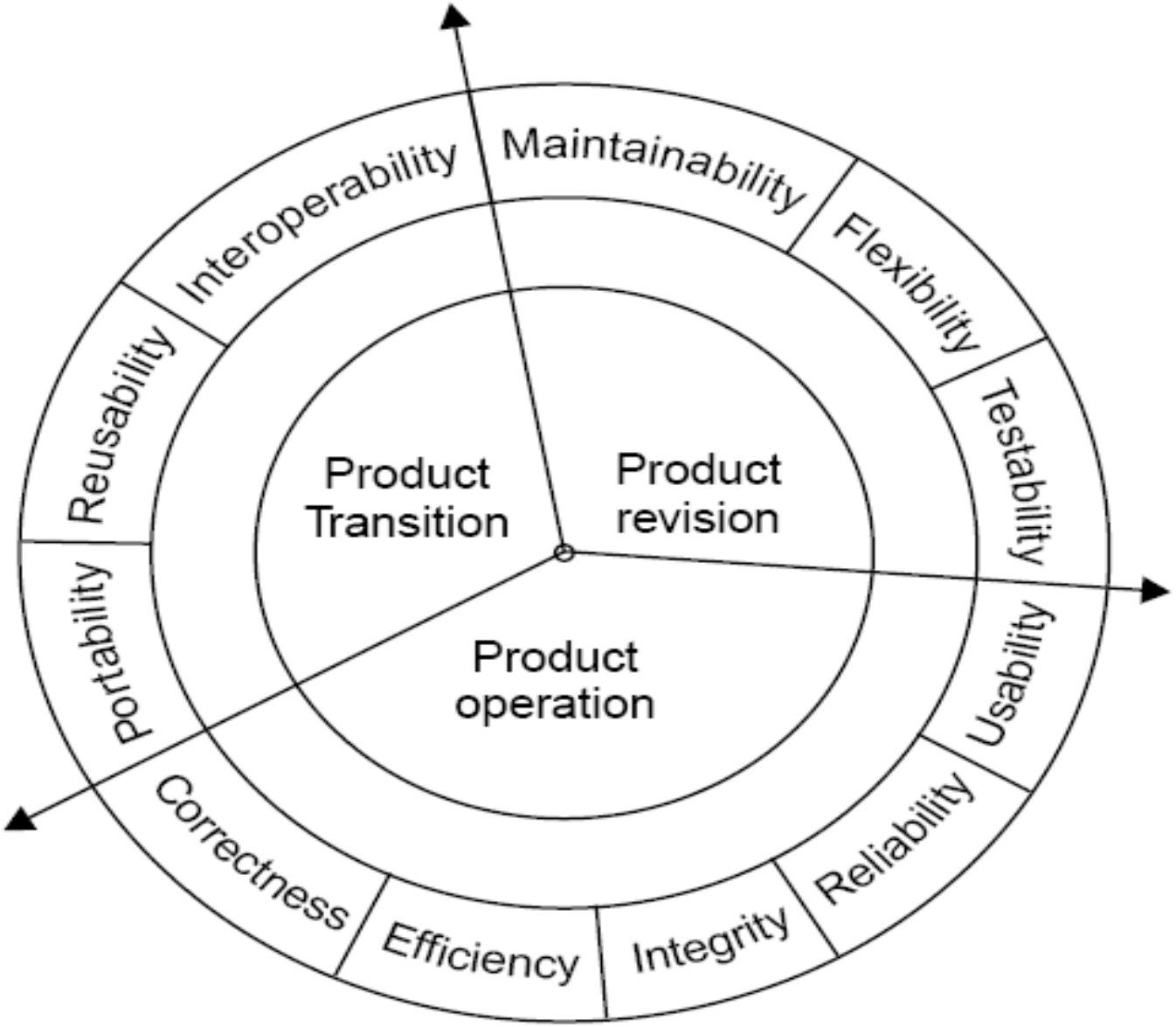
Software Quality

Different people understand different meanings of quality like:

- conformance to requirements
- fitness for the purpose
- level of satisfaction

McCall Software Quality Model

Fig 7.9: Software quality factors



Quality criteria

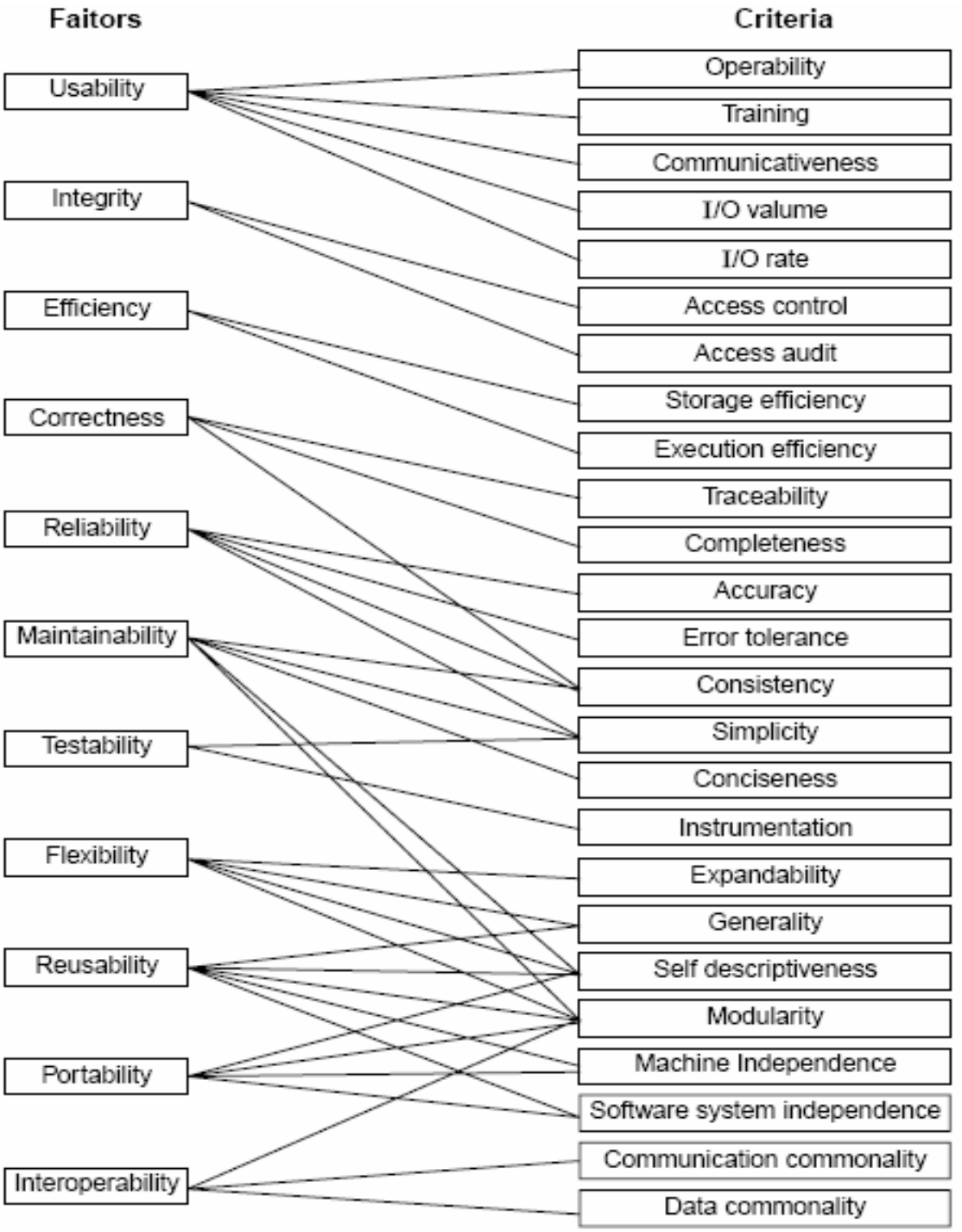


Fig 7.10: McCall's quality model

Sr. No.	*Quality Criteria*	*Usability*	*Integrity*	*Efficiency*	*Correctness*	*Reliability*	*Maintain-bility*	*Testability*	*Flexibility*	*Reusability*	*Portability*	*Interopera-bility*
1.	Operability	×										
2.	Training	×										
3.	Communicativeness	×										
4.	I/O volume	×										
5.	I/O rate	×										
6.	Access control		×									
7.	Access Audit		×									
8.	Storage efficiency			×								
9.	Execution Efficiency			×								
10.	Traceability				×							
11.	Completeness				×							
12.	Accuracy					×						
13.	Error tolerance					×						
14.	Consistency				×	×	×					
15.	Simplicity					×	×	×				
16.	Conciseness						×					
17.	Instrumentation							×				
18.	Expandability								×			
19.	Generality								×	×		
20.	Self-descriptiveness						×		×	×	×	
21.	Modularity						×		×	×	×	×
22.	Machine independence									×	×	
23.	S/W system independence									×	×	
24.	Communication commonality											×
25.	Data commonality											×

Table 7.5(a): Relation between quality factors and quality criteria

Boehm Software Quality Model

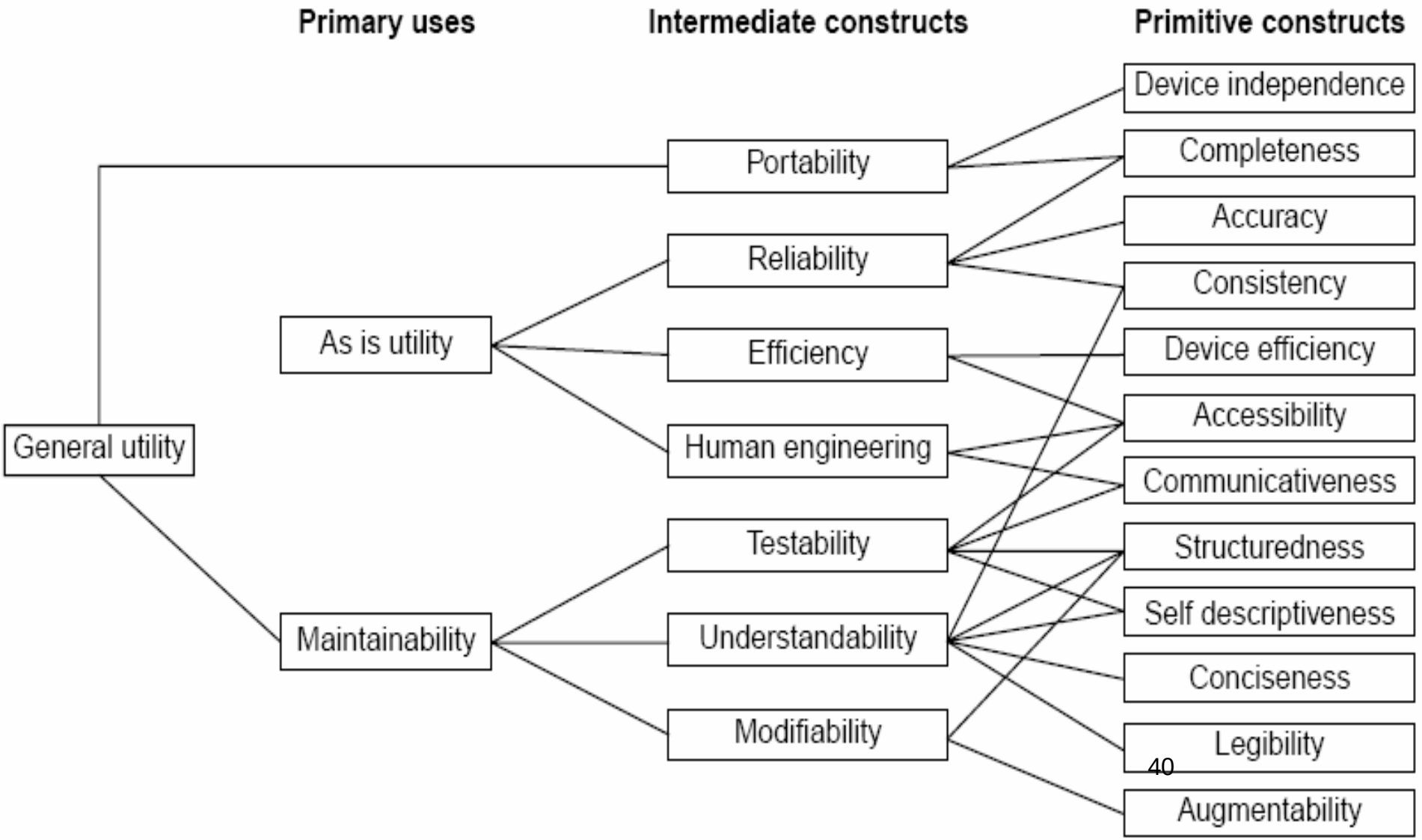


Fig.7.11: The Boehm software quality model

ISO 9126

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Characteristic/ Attribute	Short Description of the Characteristics and the concerns Addressed by Attributes
Functionality	Characteristics relating to achievement of the basic purpose for which the software is being engineered
• Suitability	The presence and appropriateness of a set of functions for specified tasks
• Accuracy	The provision of right or agreed results or effects
• Interoperability	Software's ability to interact with specified systems
• Security	Ability to prevent unauthorized access, whether accidental or deliberate, to program and data.
Reliability	Characteristics relating to capability of software to maintain its level of performance under stated conditions for a stated period of time
• Maturity	Attributes of software that bear on the frequency of failure by faults in the software

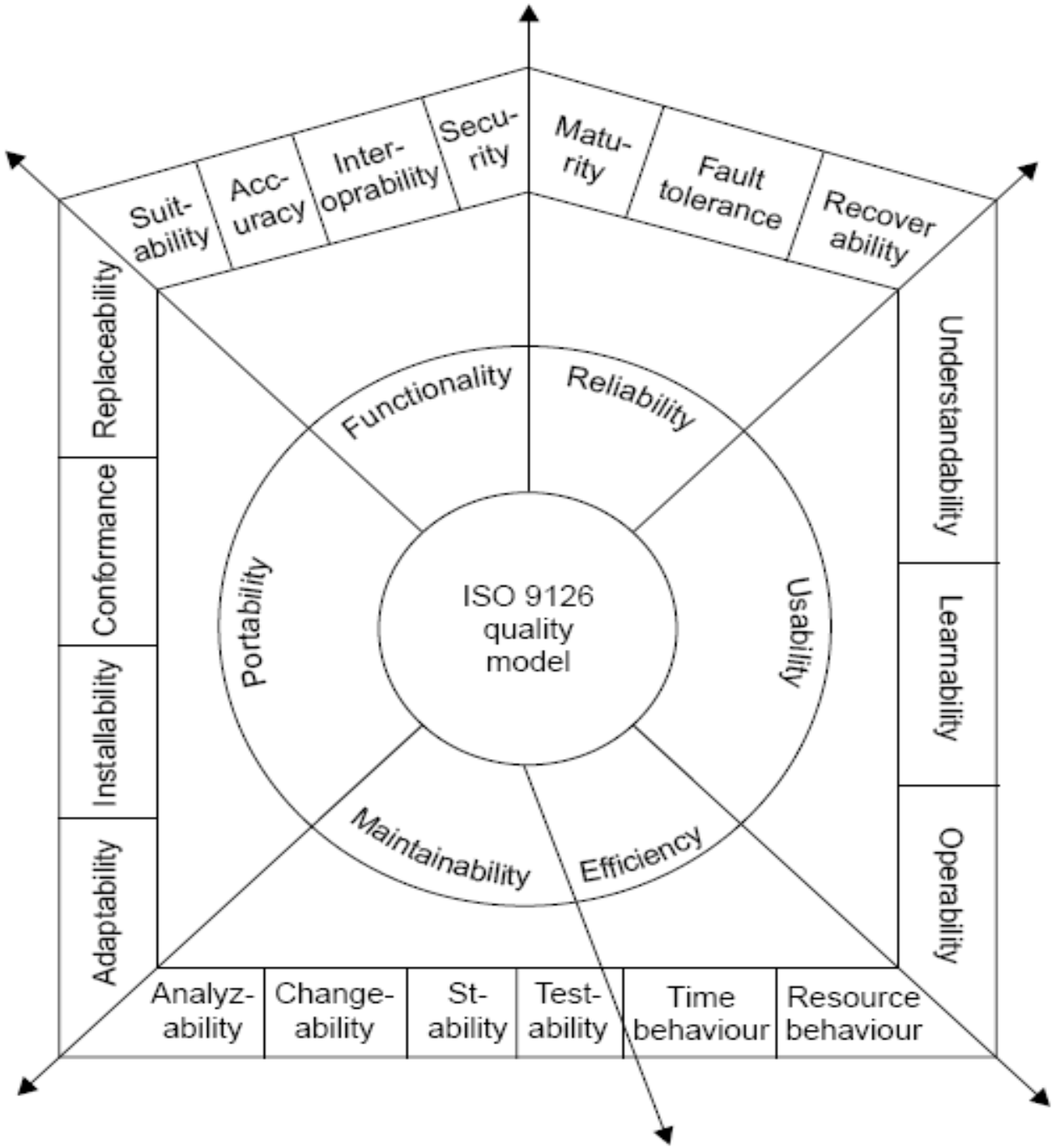
• Fault tolerance	Ability to maintain a specified level of performance in cases of software faults or unexpected inputs
• Recoverability	Capability and effort needed to reestablish level of performance and recover affected data after possible failure.
Usability	Characteristics relating to the effort needed for use, and on the individual assessment of such use, by a stated implied set of users.
• Understandability	The effort required for a user to recognize the logical concept and its applicability.
• Learnability	The effort required for a user to learn its application, operation, input and output.
• Operability	The ease of operation and control by users.
Efficiency	Characteristic related to the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

• Time behavior	The speed of response and processing times and throughout rates in performing its function.
• Resource behavior	The amount of resources used and the duration of such use in performing its function.
Maintainability	Characteristics related to the effort needed to make modifications, including corrections, improvements or adaptation of software to changes in environment, requirements and functions specifications.
• Analyzability	The effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.
• Changeability	The effort needed for modification, fault removal or for environmental change.
• Stability	The risk of unexpected effect of modifications.
• Testability	The effort needed for validating the modified software.

Portability	Characteristics related to the ability to transfer the software from one organization or hardware or software environment to another.
• Adaptability	The opportunity for its adaptation to different specified environments.
• Installability	The effort needed to install the software in a specified environment.
• Conformance	The extent to which it adheres to standards or conventions relating to portability.
• Replaceability	The opportunity and effort of using it in the place of other software in a particular environment.

Table 7.6: Software quality characteristics and attributes – The ISO 9126 view

Fig.7.12: ISO 9126 quality model



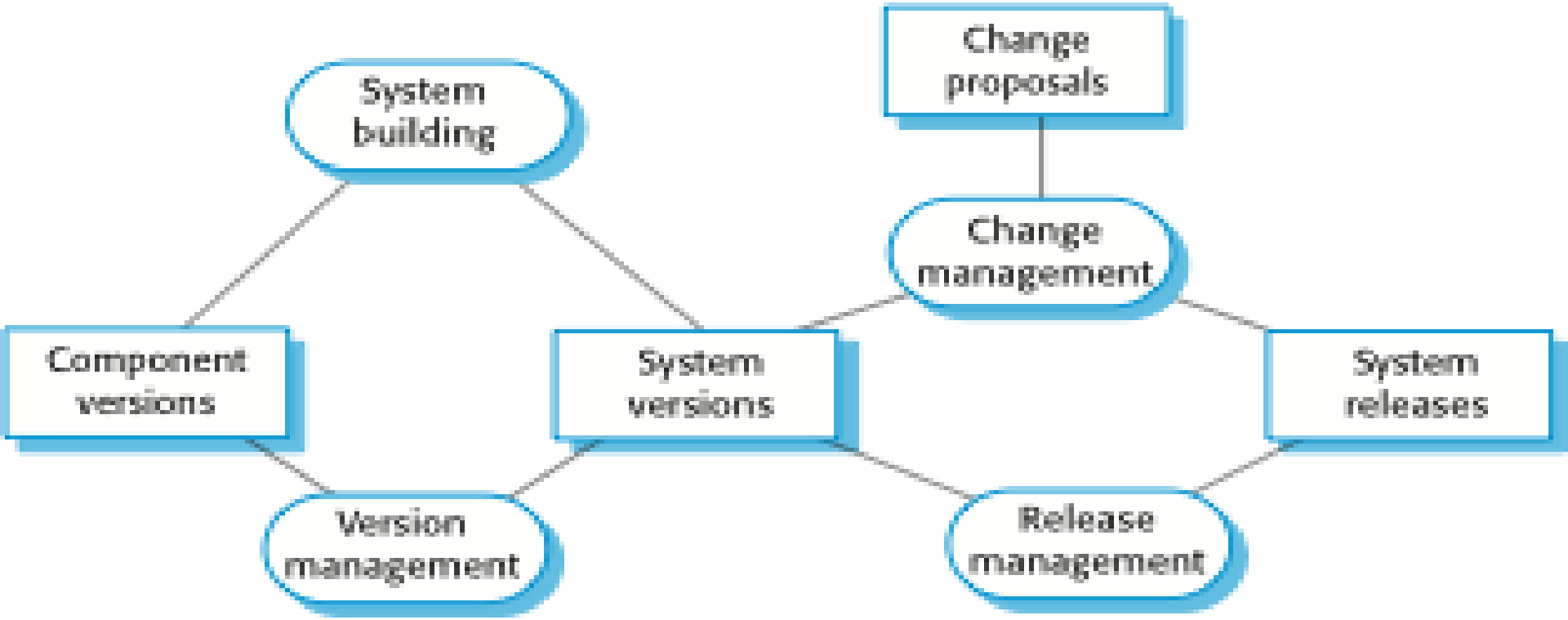
Configuration management

Configuration management (CM) is ***concerned with the policies, processes and tools for managing changing software systems.***

CM activities

- **Change management**: Keeping track of requests for changes, working out the costs and impact of changes, and deciding the changes
- **Version management;** Keeping track of the multiple versions of system components.
- **System building:** The process of assembling program components, data and libraries, then compiling these to create an executable system.
- **Release management:** Preparing software for external release and keeping track of the system versions that have been released.

CM activities



CM terminology

Term	Explanation
Configuration item or software configuration item (SCI)	**Anything associated with a software project** (design, code, test data, document, etc.) **that has been placed under configuration control**. There are often different versions of a configuration item. Configuration items have a unique name.
Configuration control	**The process of ensuring that versions of systems and components are recorded and maintained** so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Version	**An instance of a configuration item that differs**, in some way, **from other instances of that item**. **Versions always have a unique identifier**, which is often composed of the **configuration item name plus a version number**.
Baseline	**A baseline is a collection of component versions that make up a system**.
Codeline	**A codeline is a set of versions of a software component and other configuration items on which that component depends.**

CM terminology

Term	Explanation
Mainline	**A sequence of baselines representing different versions of a system.**
Release	**A version of a system that has been released to customers** (or other users in an organization) for use.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.
Branching	**The creation of a new codeline from a version in an existing codeline.** The new codeline and the existing codeline may then develop independently.
Merging	**The creation of a new version of a software component by merging separate versions in different codelines**. These codelines may have been created by a previous branch of one of the codelines involved.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.

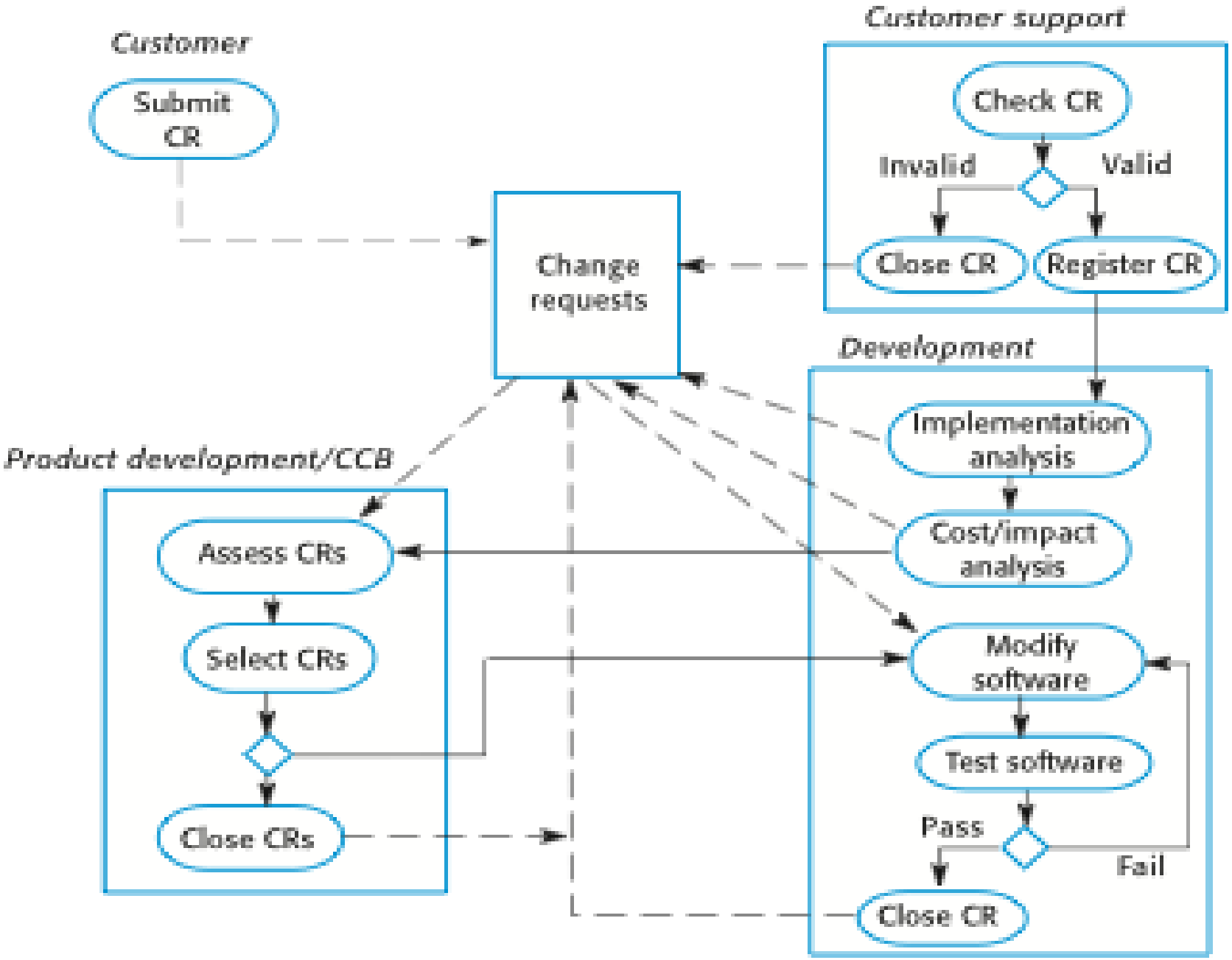
Change management

- **Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.**

Factors in change analysis

- The **consequences of not making the change**
- The **benefits of the change**
- The **number of users affected** by the change
- The **costs of making the change**
- The **product release cycle**

The change management process



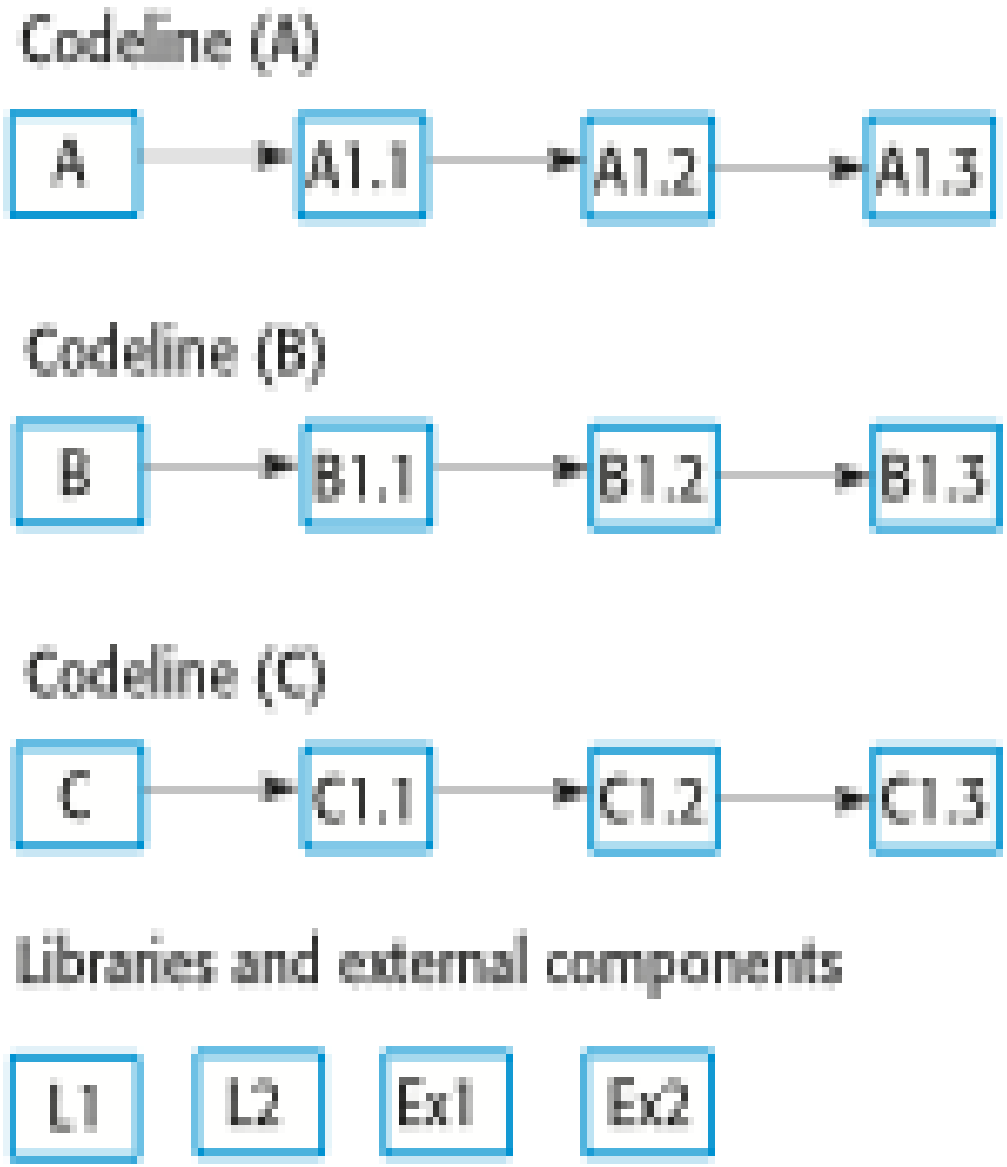
Version management

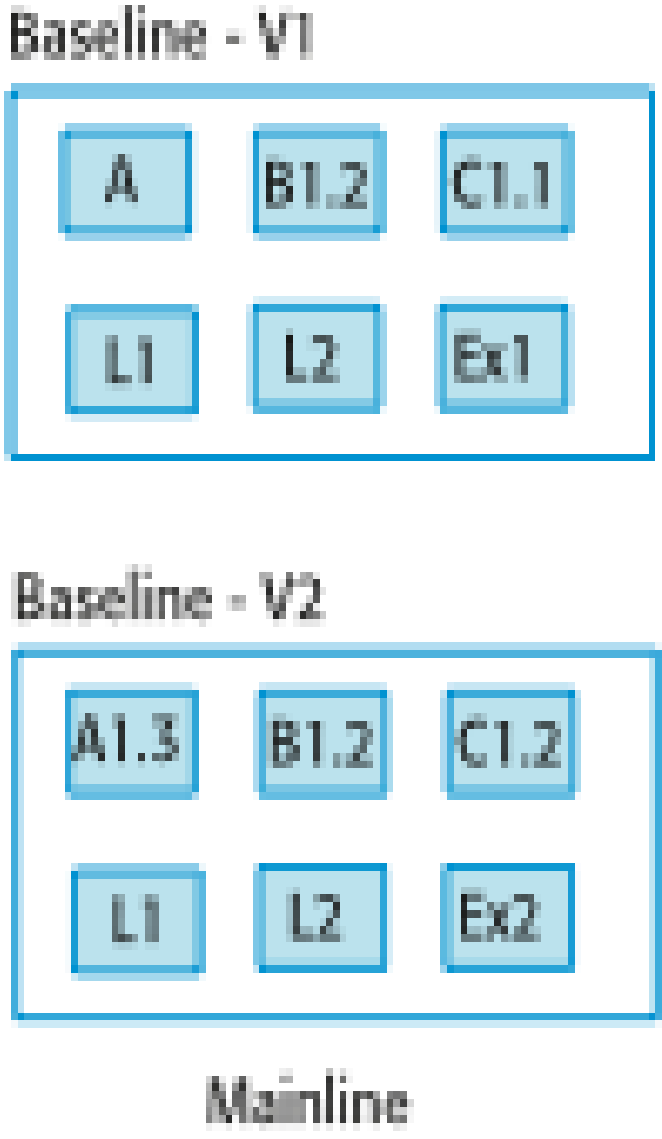
Therefore **version management can be thought of as the process of managing codelines and baselines.**

Codelines and baselines

- A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- **Codelines normally apply to components of systems** so that there are **different versions of each component.**
- **A baseline is a definition of a specific system**.
- **The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.**

Codelines and baselines

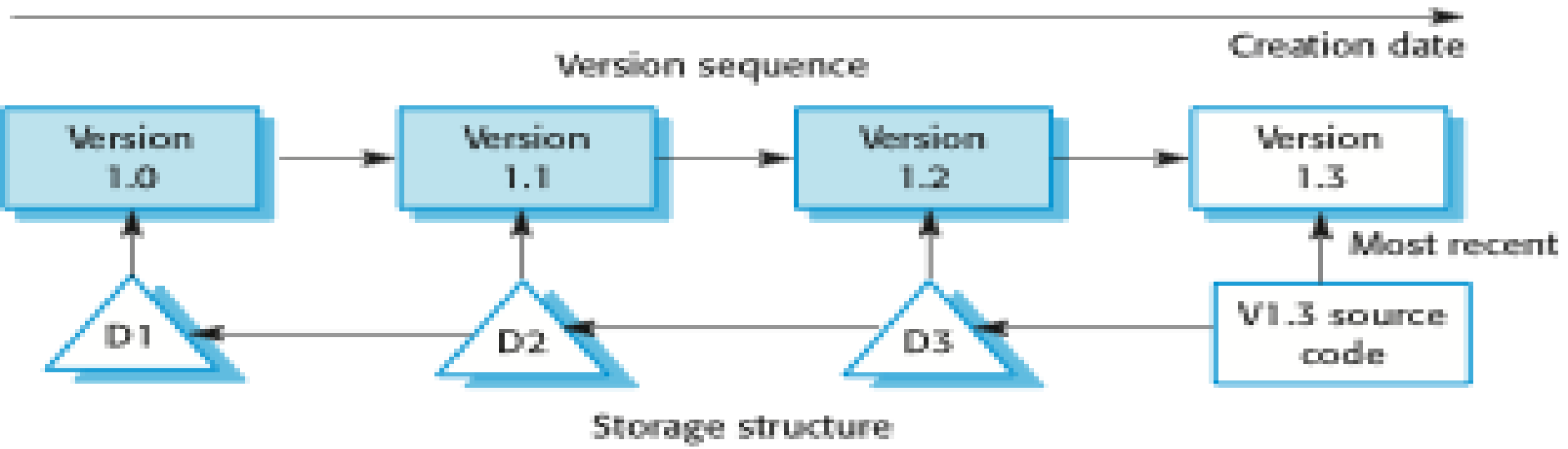




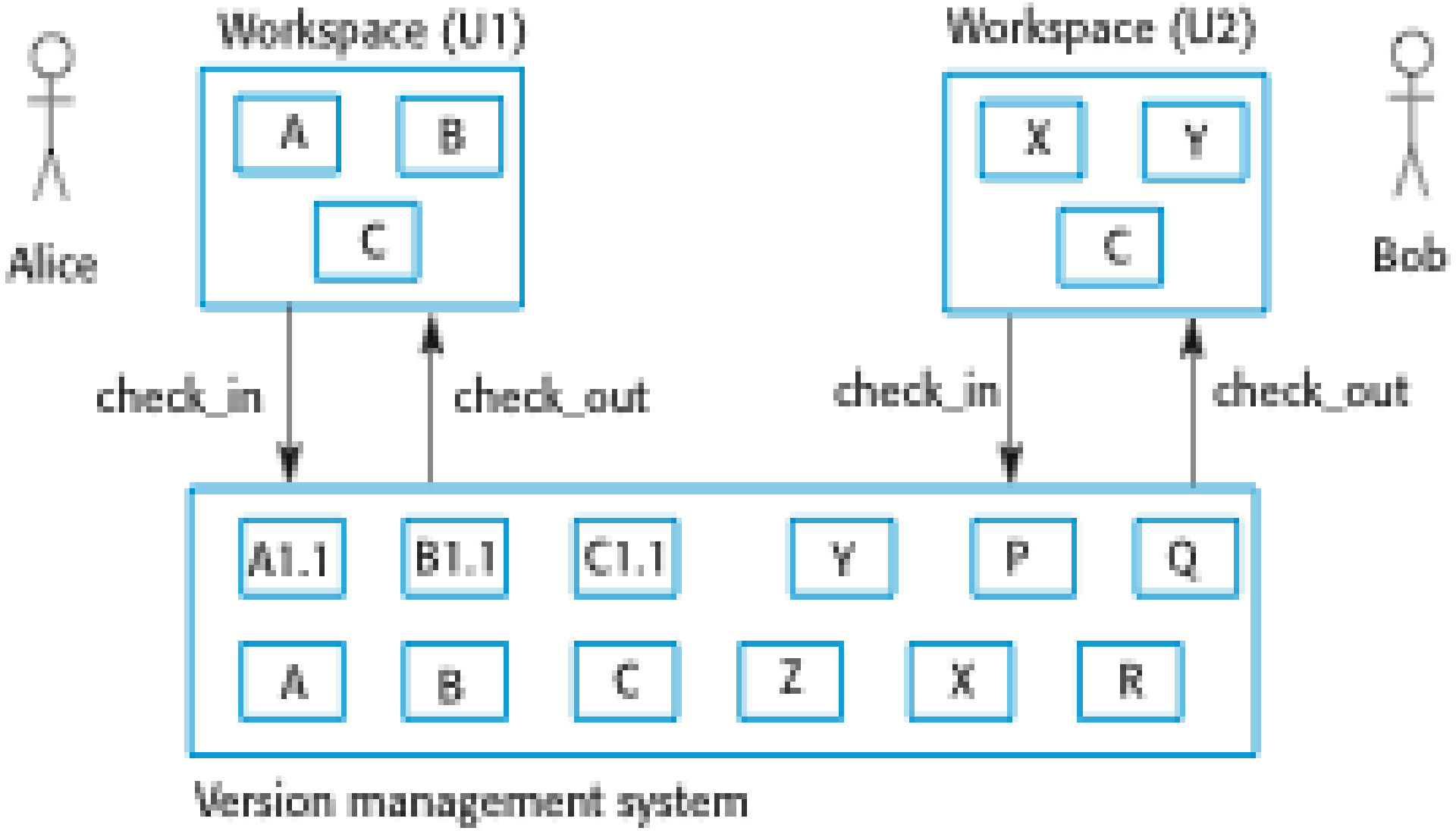
Version management systems

- **Version and release identification:** Managed versions are assigned identifiers
- **Storage management**: To reduce the storage space required by multiple versions of components that differ only slightly,
- **Change history recording**: changes made to the code of a system or component are recorded.
- **Independent development**: ensures that changes made to a component by different developers do not interfere.
- **Project support**: support the development of several projects, which share components.

Storage management using deltas



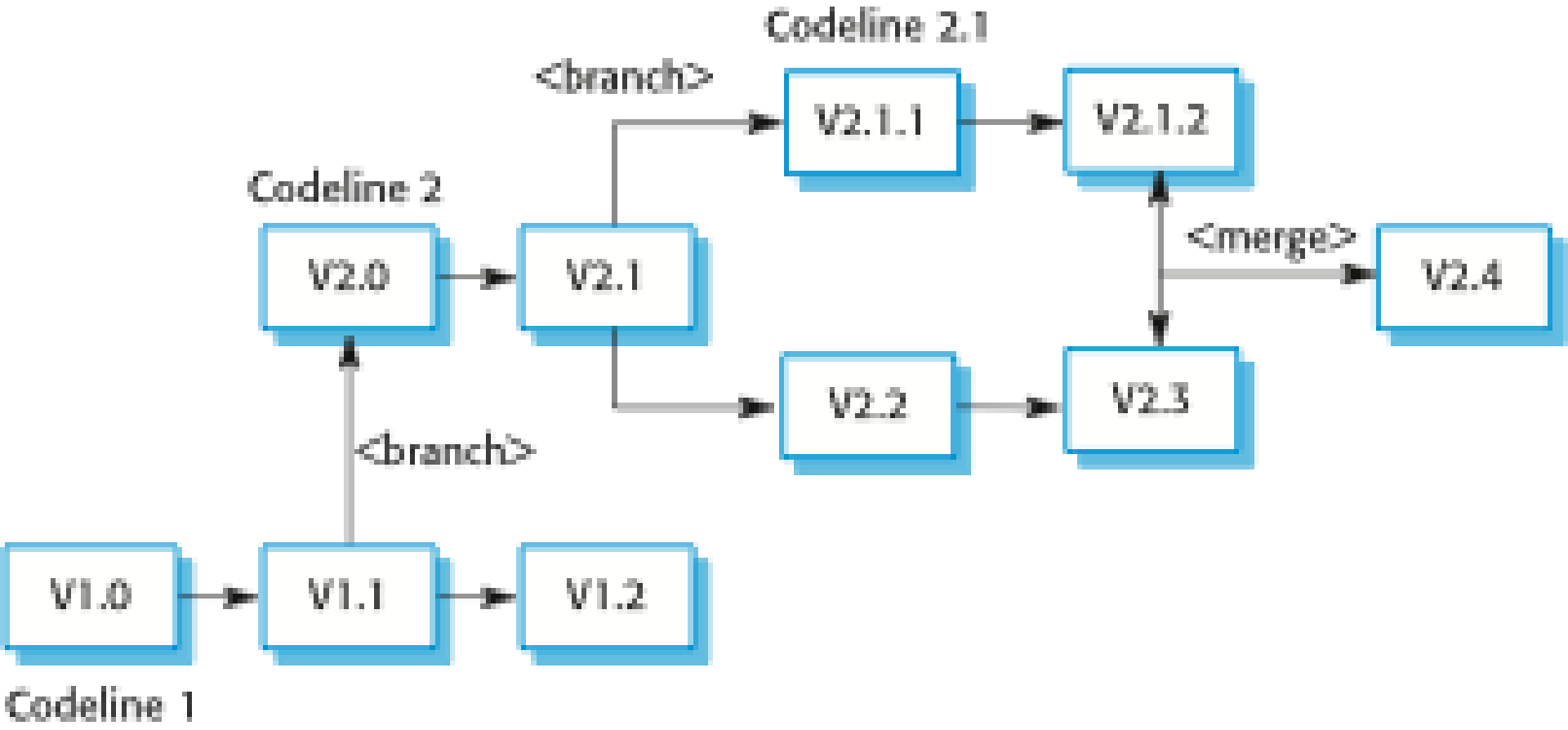
Check-in and check-out from a version repository



Codeline branches

- Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.
  - This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- **At some stage**, **it may be necessary to merge codeline branches to create a new version of a component** that includes all changes that have been made.
  - If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.

Branching and merging



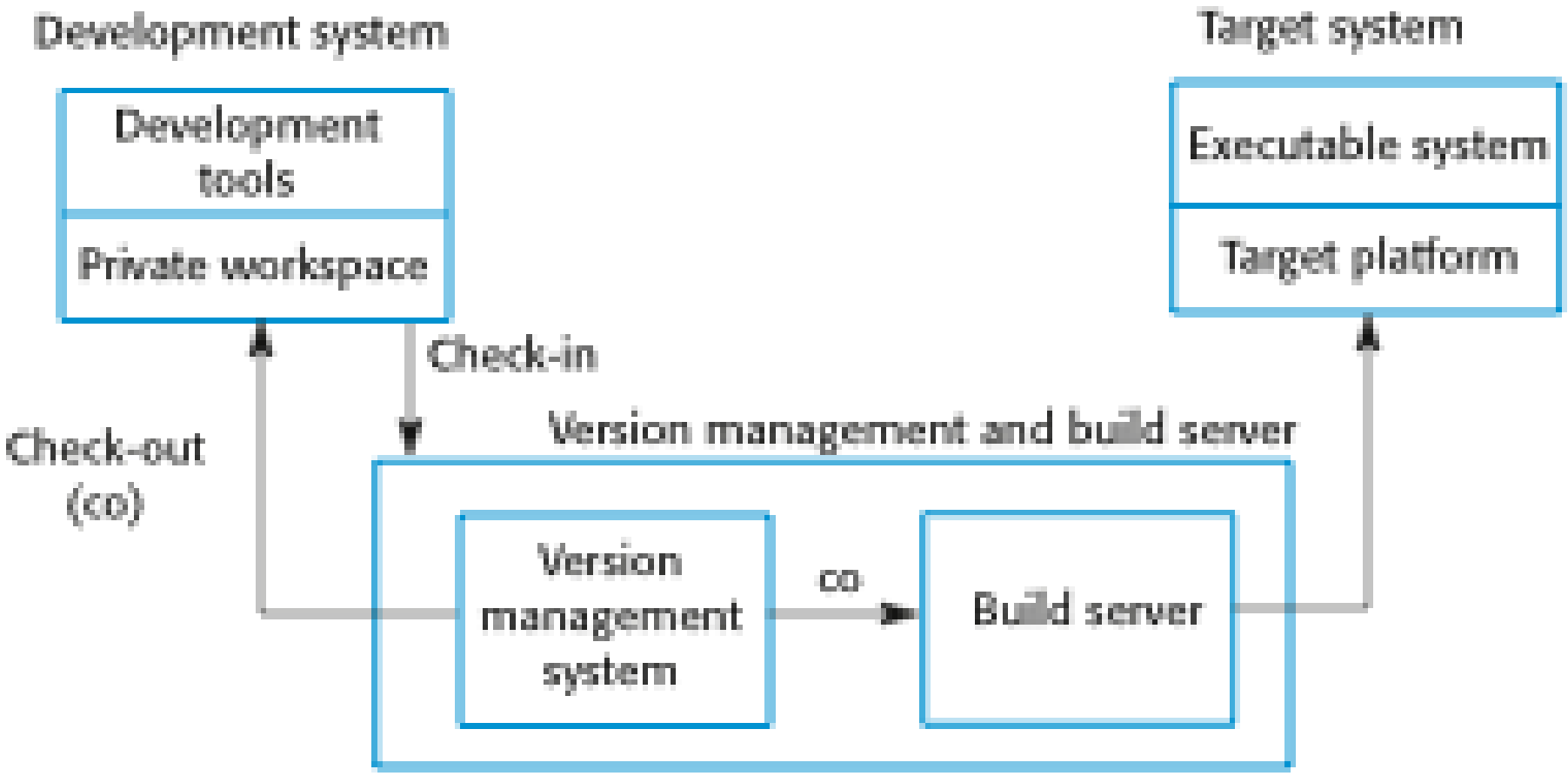
System building

- System building is the **process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files,** etc.

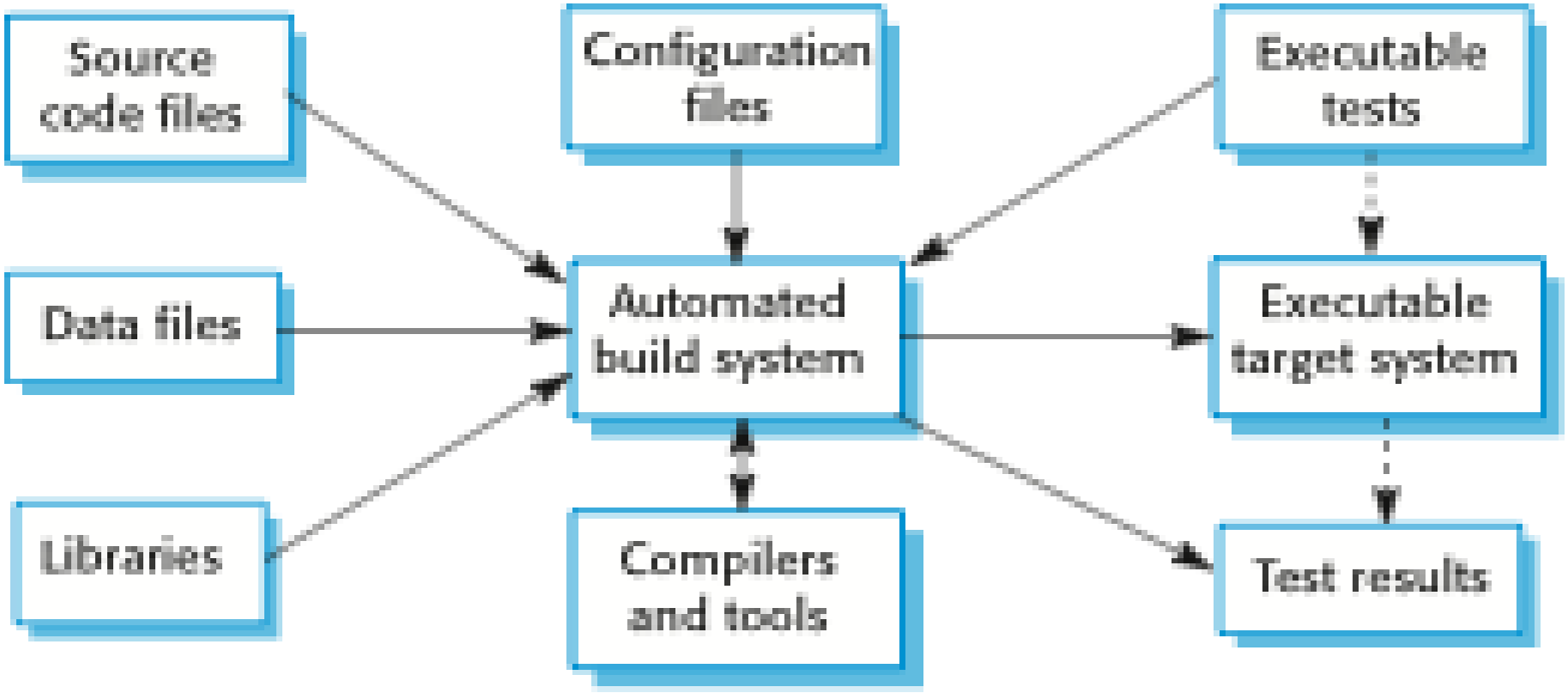
Build platforms

- **The development system, which includes development tools such as compilers, source code editors**, etc.
  - Developers check out code from the version management system into a private workspace before making changes to the system.
- **The build server, which is used to build definitive, executable versions of the system.**
  - Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.
- **The target environment, which is the platform on which the system executes.**

Development, build, and target platforms



System building



Release management

- A system release is **a version of a software system that is distributed to customers**.
- For mass market software, it is usually possible to identify **two types** of release: **major releases** which **deliver significant new functionality**, and **minor releases**, which **repair bugs and fix customer problems** that have been reported.

Release tracking

- **In the event of a problem**, it may be necessary to **reproduce exactly the software that has been delivered** to a particular customer.
- When a **system release** is produced, it **must be documented** to ensure that it can be re-created exactly in the future.

Release reproduction

- **To document a release**, you have to **record the specific versions of the source code components** that were **used to create the executable code -** You must **keep copies of the source code files**, **corresponding executables** and all data and configuration files - You should also **record the versions of the operating system, libraries, compilers and other tools** used to build the software.

Release components

- **As well as the the executable code, a release may also include**:
  - **configuration files** defining how the release should be configured for particular installations;
  - **data files**, such as files of error messages, that are needed for successful system operation;
  - **an installation program** that is used to help install the system on target hardware;
  - **electronic and paper documentation** describing the system;
  - **packaging** and associated publicity that have been designed for that release.