SWEN 6301 Software Construction

Module 9: Software Evolution and Configuration Management

Ahmed Tamrawi

Software Evolution

Topics Covered

- Evolution processes
- Legacy systems
- Software maintenance

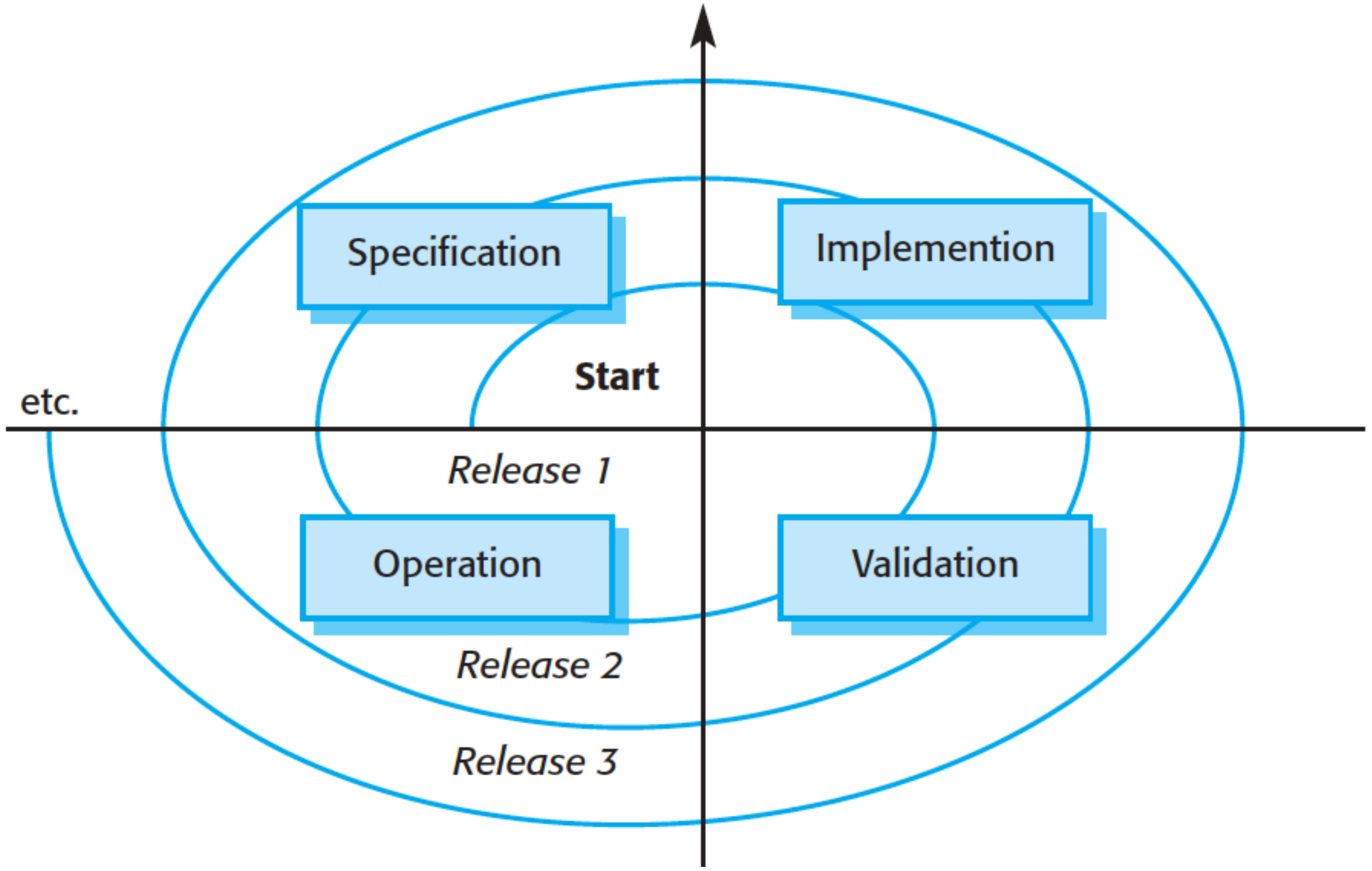
Software Change

- Software change is **inevitable**
  - New requirements emerge when the software is used;
  - The business environment changes;
  - Errors must be repaired;
  - New computers and equipment is added to the system;
  - The performance or reliability of the system may have to be improved.
- A key problem for all organizations is **implementing and managing change** to their existing software systems.

Importance of Evolution

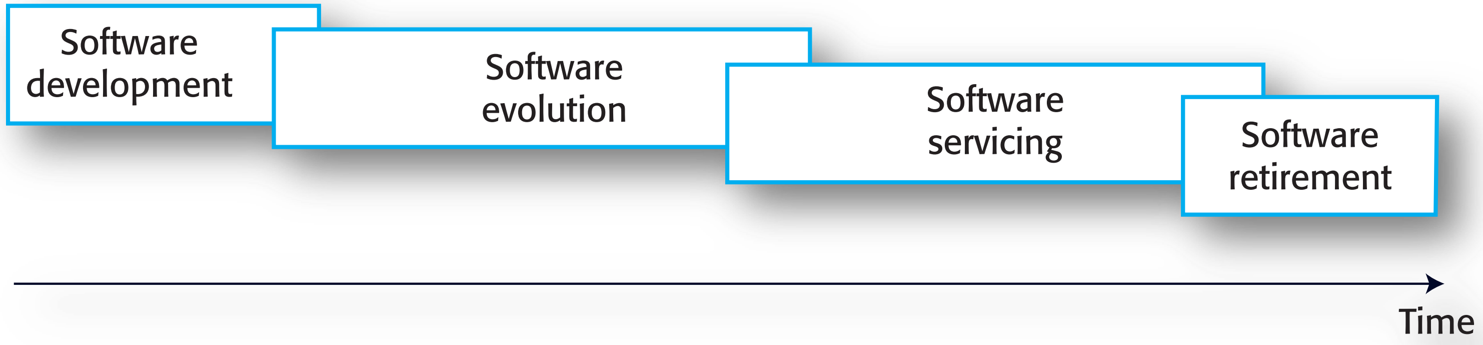
- Organizations have huge investments in their software systems - they are **critical business assets**.
- To maintain the value of these assets to the business, they must be **changed and updated**.
- The majority of the software budget in large companies is devoted to **changing and evolving existing software rather than developing new software**.

Spiral Model of Development and Evolution



Evolution and Servicing

- **Evolution**
  - The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.
- **Servicing**
  - At this stage, the software remains useful but **the only changes made are those required to keep it operational** i.e. bug fixes and changes to reflect changes in the software's environment. <u>No new functionality is added</u>.
- **Phase-out**
  - The software may still be used but no further changes are made to it.

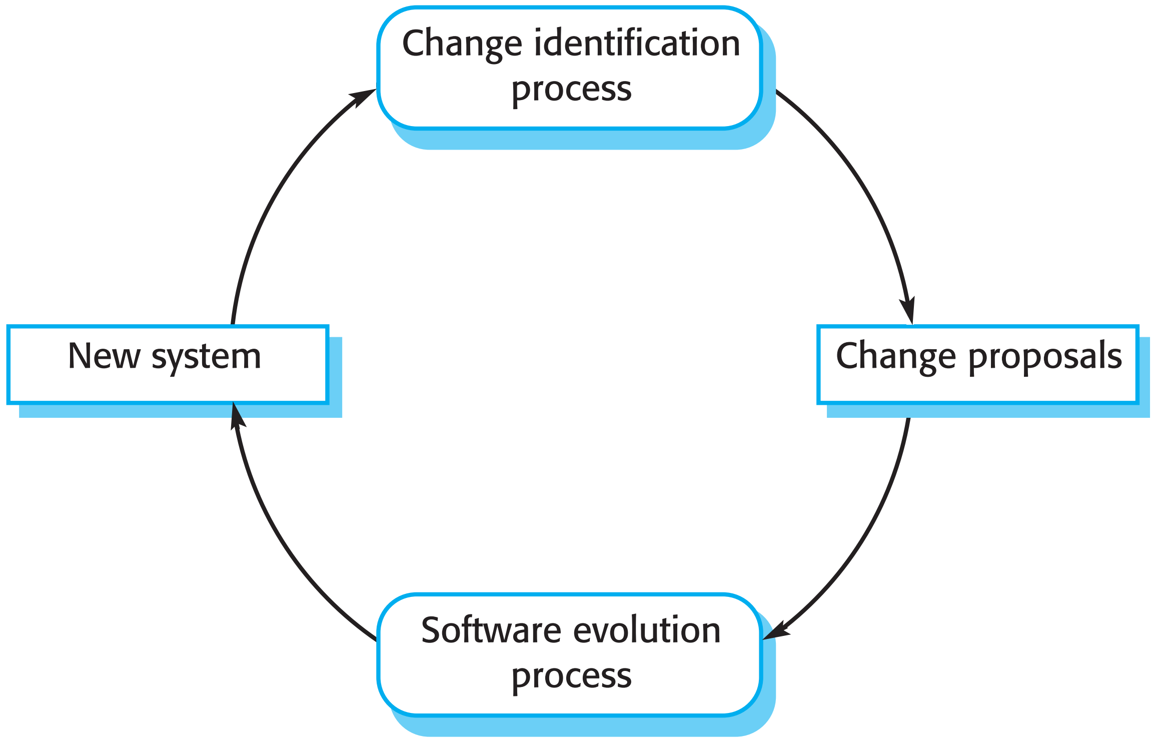


Evolution Processes

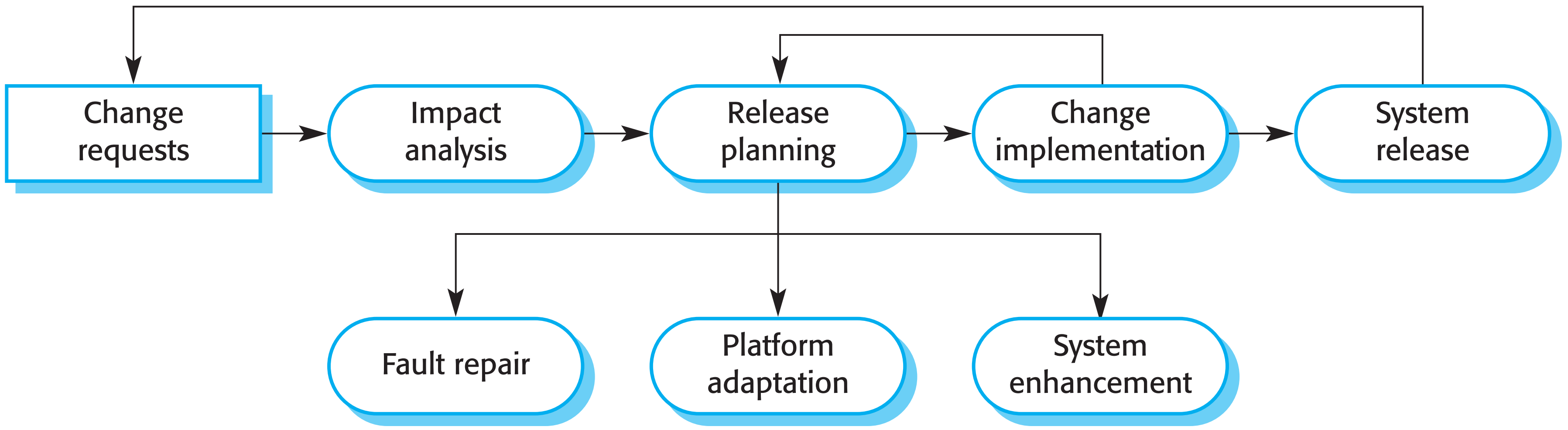
Evolution Processes

- Software evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- **Proposals for change are the driver for system evolution**.
  - Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- **Change identification** and **evolution** continues throughout the system lifetime.

Change Identification and Evolution Processes

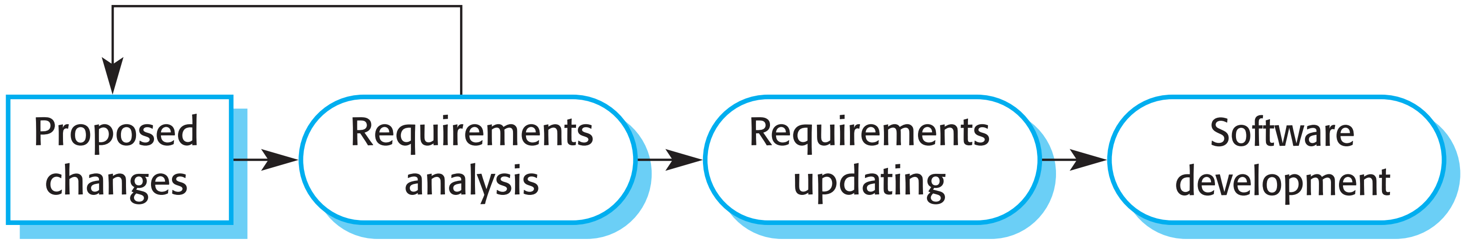


Software Evolution Process



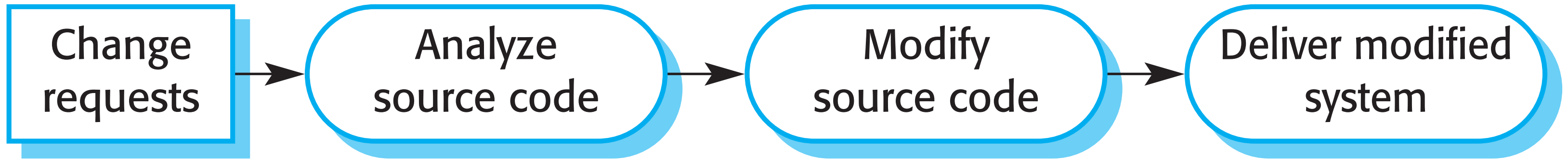
Change Implementation

- Iteration of the development process where the revisions to the system are **designed**, **implemented** and **tested**.
- A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
- During the program understanding phase, you have to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.



Urgent Change Requests

- Urgent changes may have to be implemented without going through all stages of the software engineering process
  - If a serious system fault has to be repaired to allow normal operation to continue;
  - If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
  - If there are business changes that require a very rapid response (e.g. the release of a competing product).



Agile Methods and Evolution

- Agile methods are based on **incremental development** so the transition from development to evolution is a seamless one.
  - Evolution is simply a continuation of the development process based on frequent system releases.
- **Automated regression testing** is particularly valuable when changes are made to a system.
- Changes may be **expressed as additional user stories**.

Handover Problems

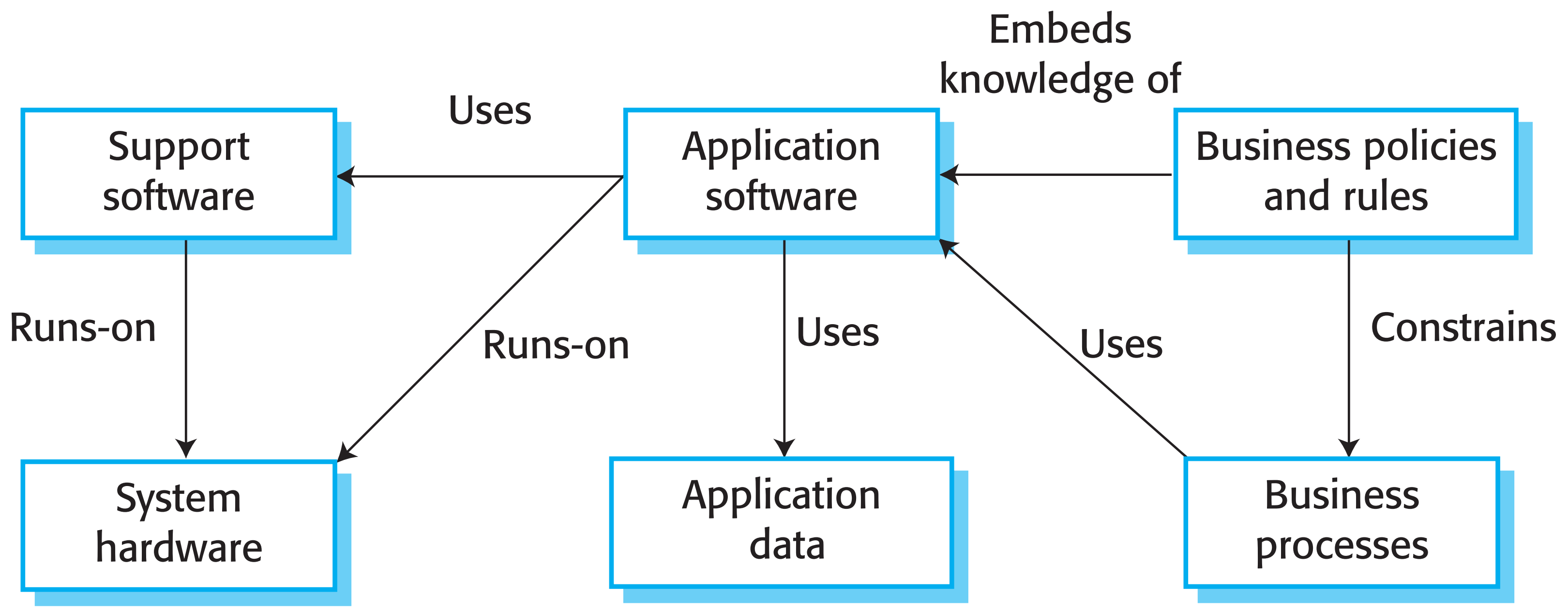
- Where the development team have used an agile approach but the evolution team is unfamiliar with agile methods and prefer a plan-based approach.
  - The evolution team may expect **detailed documentation to support evolution and this is not produced in agile processes**.
- Where a plan-based approach has been used for development but the evolution team prefer to use agile methods.
  - The evolution team may have to start from scratch developing automated tests and the code in the system may not have been re-factored and simplified as is expected in agile development.

Legacy Systems

Legacy Systems

- Legacy systems are **older systems that rely on languages and technology that are no longer used for new systems development**.
- Legacy software may be **dependent on older hardware**, such as mainframe computers and may have associated legacy processes and procedures.
- Legacy systems are not just software systems but are broader socio-technical systems that include hardware, software, libraries and other supporting software and business processes.

Elements of a Legacy System



Legacy System Components

- *System hardware* Legacy systems may have been written for hardware that is no longer available.
- *Support software* The legacy system may rely on a range of support software, which may be obsolete or unsupported.
- *Application software* The application system that provides the business services is usually made up of a number of application programs.
- *Application data* These are data that are processed by the application system. They may be inconsistent, duplicated or held in different databases.

Legacy System Components

- *Business processes* These are processes that are used in the business to achieve some business objective.
- Business processes may be designed around a legacy system and constrained by the functionality that it provides.
- *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

Legacy System Replacement

- Legacy system replacement is **risky** and **expensive** so businesses continue to use these systems
- System replacement is risky for a number of reasons
  - Lack of complete system specification
  - Tight integration of system and business processes
  - Undocumented business rules embedded in the legacy system
  - New software development may be late and/or over budget

Legacy System Change

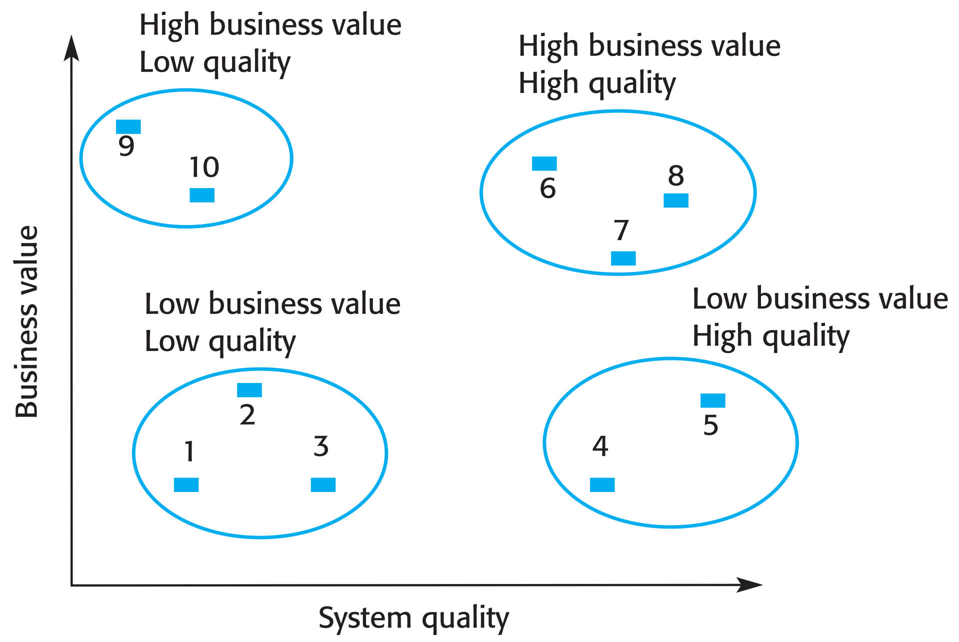
- Legacy systems are **expensive** to change for a number of reasons:
  - No consistent programming style
  - Use of obsolete programming languages with few people available with these language skills
  - Inadequate system documentation
  - System structure degradation
  - Program optimizations may make them hard to understand
  - Data errors, duplication and inconsistency

Legacy System Management

- Organisations that rely on legacy systems must choose a strategy for evolving these systems
    - Scrap the system completely and modify business processes so that it is no longer required;
    - Continue maintaining the system;
    - Transform the system by re-engineering to improve its maintainability;
    - Replace the system with a new system.
- The strategy chosen should depend on the **system quality** and its **business value**.

Example of a Legacy System Assessment

- Low quality, low business value
  - These systems should be scrapped.
- Low-quality, high-business value
  - These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.
- High-quality, low-business value
  - Replace with COTS, scrap completely or maintain.
- High-quality, high business value
  - Continue in operation using normal system maintenance.



System Measurement

- You may collect quantitative data to make an assessment of the quality of the application system
  - The number of system change requests; The higher this accumulated value, the lower the quality of the system.
  - The number of different user interfaces used by the system; The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.
  - The volume of data used by the system. As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data.
  - Cleaning up old data is a very expensive and time-consuming process

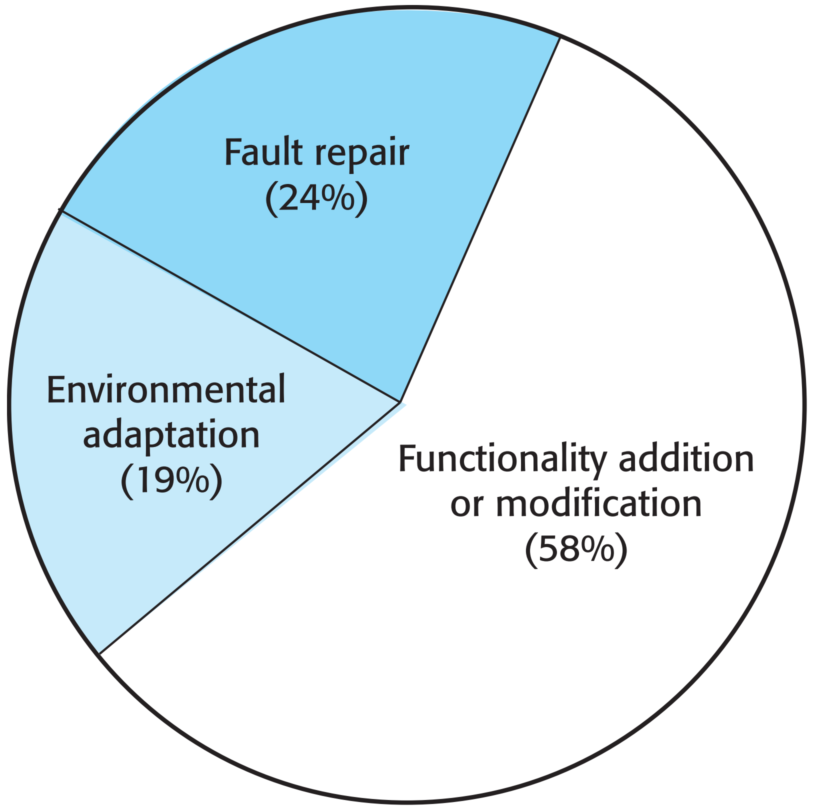
Software Maintenance

Software Maintenance

- Modifying a program after it has been put into use.
- The term is mostly used for **changing custom software**. Generic software products are said to **evolve to create new versions**.
- Maintenance does not normally involve major changes to the system's architecture.
- Changes are implemented by modifying existing components and adding new components to the system.

Types of Maintenance

- **Fault repairs**
  - Changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements.
- **Environmental adaptation**
  - Maintenance to adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- **Functionality addition and modification**
  - Modifying the system to satisfy new requirements.



Maintenance Costs

- **Usually greater than development costs** (2* to 100* depending on the application).
- Affected by both **technical and non-technical** factors.
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- **Ageing software can have high support costs** (e.g. old languages, compilers etc.).

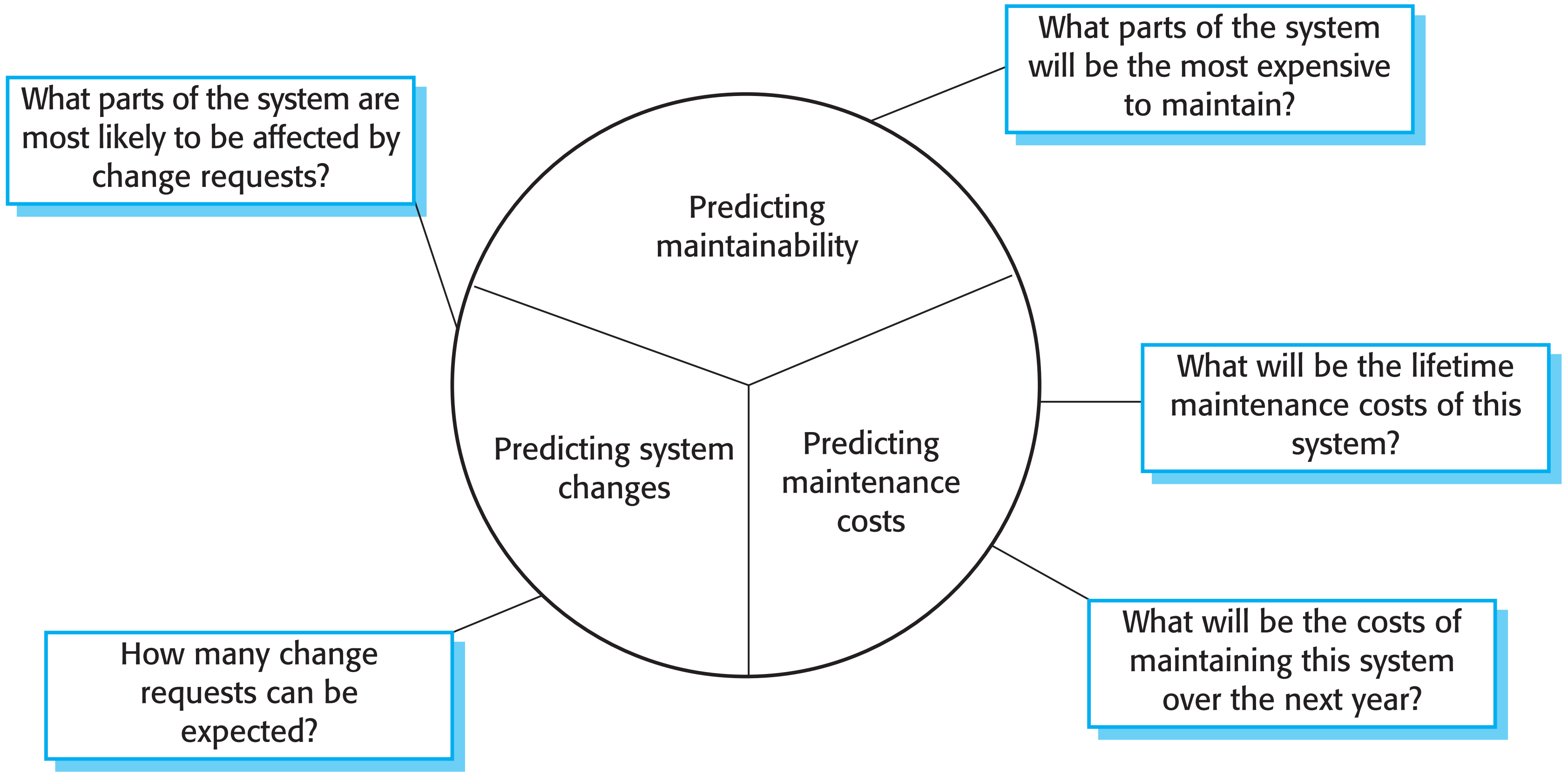
Maintenance Costs

- It is usually more expensive to add new features to a system during maintenance than it is to add the same features during development
  - A new team has to understand the programs being maintained
  - Separating maintenance and development means there is no incentive for the development team to write maintainable software
  - Program maintenance work is unpopular
    - Maintenance staff are often inexperienced and have limited domain knowledge.
  - As programs age, their structure degrades and they become harder to change

Maintenance Prediction

- **Maintenance prediction** is concerned with assessing which parts of the system may cause problems and have high maintenance costs
    - Change acceptance depends on the maintainability of the components affected by the change;
    - Implementing changes degrades the system and reduces its maintainability;
    - Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Maintenance Prediction


What parts of the system are most likely to be affected by change requests?
What parts of the system will be the most expensive to maintain?
Predicting maintainability
Predicting system changes
Predicting maintenance costs
What will be the lifetime maintenance costs of this system?
How many change requests can be expected?
What will be the costs of maintaining this system over the next year?

Change Prediction

- Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- Tightly coupled systems require changes whenever the environment is changed.
- Factors influencing this relationship are
  - Number and complexity of system interfaces;
  - Number of inherently volatile system requirements;
  - The business processes where the system is used.

Complexity Metrics

- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that **most maintenance effort is spent on a relatively small number of system components**.
- Complexity depends on
  - Complexity of control structures;
  - Complexity of data structures;
  - Object, method (procedure) and module size.

Process Metrics

- Process metrics may be used to assess maintainability
  - Number of requests for corrective maintenance;
  - Average time required for impact analysis;
  - Average time taken to implement a change request;
  - Number of outstanding change requests.
- If any or all of these is increasing, this may indicate a decline in maintainability.

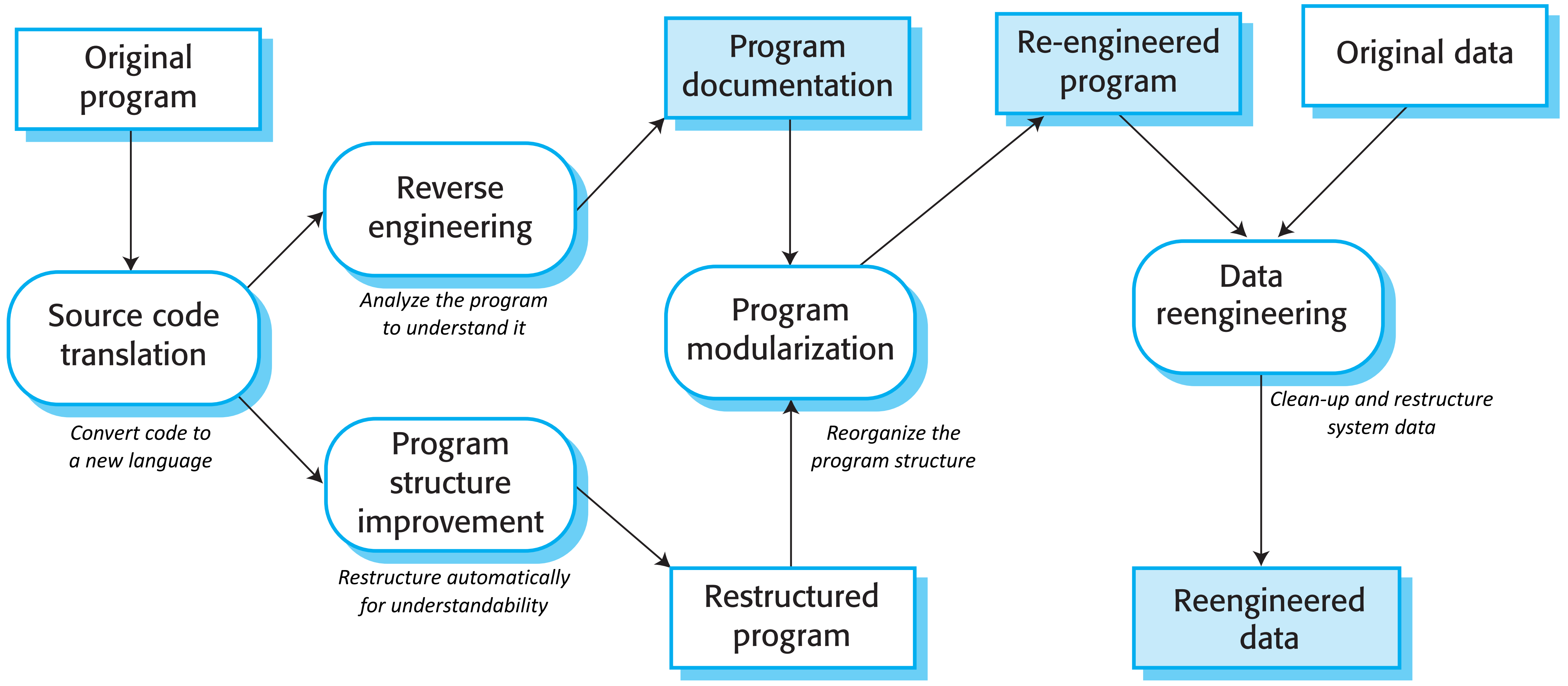
Software Reengineering

- **Restructuring or rewriting part or all of a legacy system without changing its functionality**.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Reengineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

Advantages of Reengineering

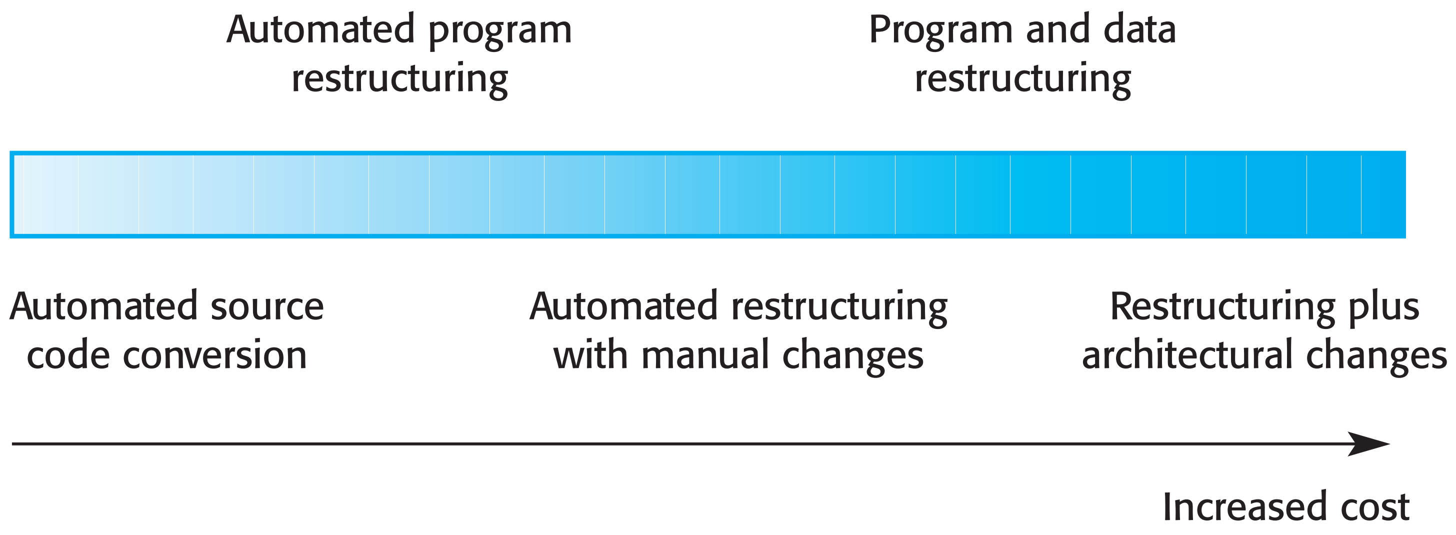
- **Reduced risk**
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- **Reduced cost**
  - The cost of re-engineering is often significantly less than the costs of developing new software.

The Reengineering Process



Original program
Source code translation
Convert code to a new language
Reverse engineering
Analyze the program to understand it
Program structure improvement
Restructure automatically for understandability
Program documentation
Program modularization
Reorganize the program structure
Restructured program
Re-engineered program
Original data
Data reengineering
Clean-up and restructure system data
Reengineered data

Reengineering Approaches


Automated program restructuring
Program and data restructuring
Automated source code conversion
Automated restructuring with manual changes
Restructuring plus architectural changes
Increased cost

Reengineering Cost Factors

- The quality of the software to be reengineered.
- The tool support available for reengineering.
- The extent of the data conversion which is required.
- The availability of expert staff for reengineering.
  - This can be a problem with old systems based on technology that is no longer widely used.

Refactoring

- Refactoring is the **process of making improvements to a program to slow down degradation through change**.
- You can think of refactoring as ‘**preventative maintenance**’ that reduces the problems of future change.
- Refactoring involves **modifying a program to improve its structure**, **reduce its complexity or make it easier to understand**.
- When you re-factor a program, you should **not** add functionality but rather **concentrate on program improvement**.

Refactoring and Reengineering

- Re-engineering **takes place after a system has been maintained for some time and maintenance costs are increasing**. You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.
- Refactoring is a **continuous process of improvement throughout the development and evolution process**. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Configuration Management

Topics Covered

- Version management
- System building
- Change management
- Release management

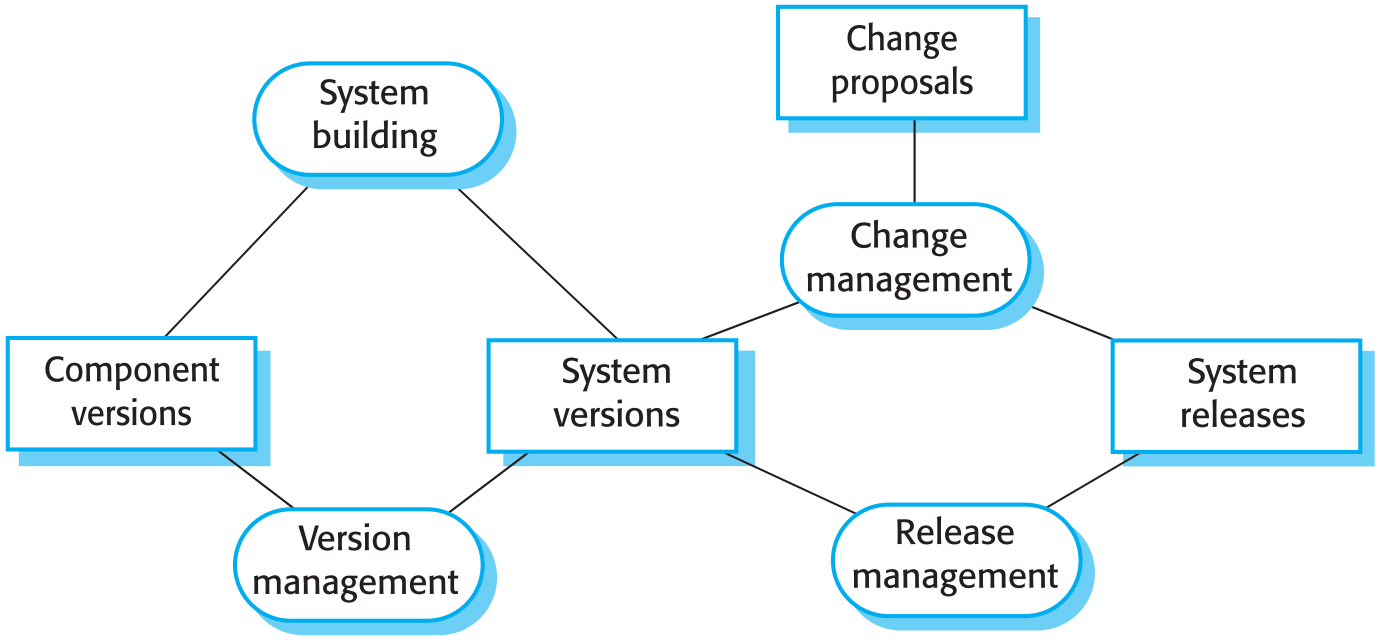
Configuration Management

- Software systems are **constantly changing** during development and use.
- **Configuration management** (CM) is concerned with the policies, processes and tools for managing changing software systems.
- You need CM because it is easy to **lose track of what changes and component versions have been incorporated into each system version**.
- CM is **essential** for team projects to control changes made by different developers

Configuration Management Activities

- **Version management**
  - Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- **System building**
  - The process of assembling program components, data and libraries, then compiling these to create an executable system.
- **Change management**
  - Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- **Release management**
  - Preparing software for external release and keeping track of the system versions that have been released for customer use.

Configuration Management Activities



Agile Development and CM

- Agile development, where components and systems are changed several times per day, is **impossible** without using CM tools.
- The definitive versions of components are held in a **shared project repository** and developers copy these into their own workspace.
- They make changes to the code then use system building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository.

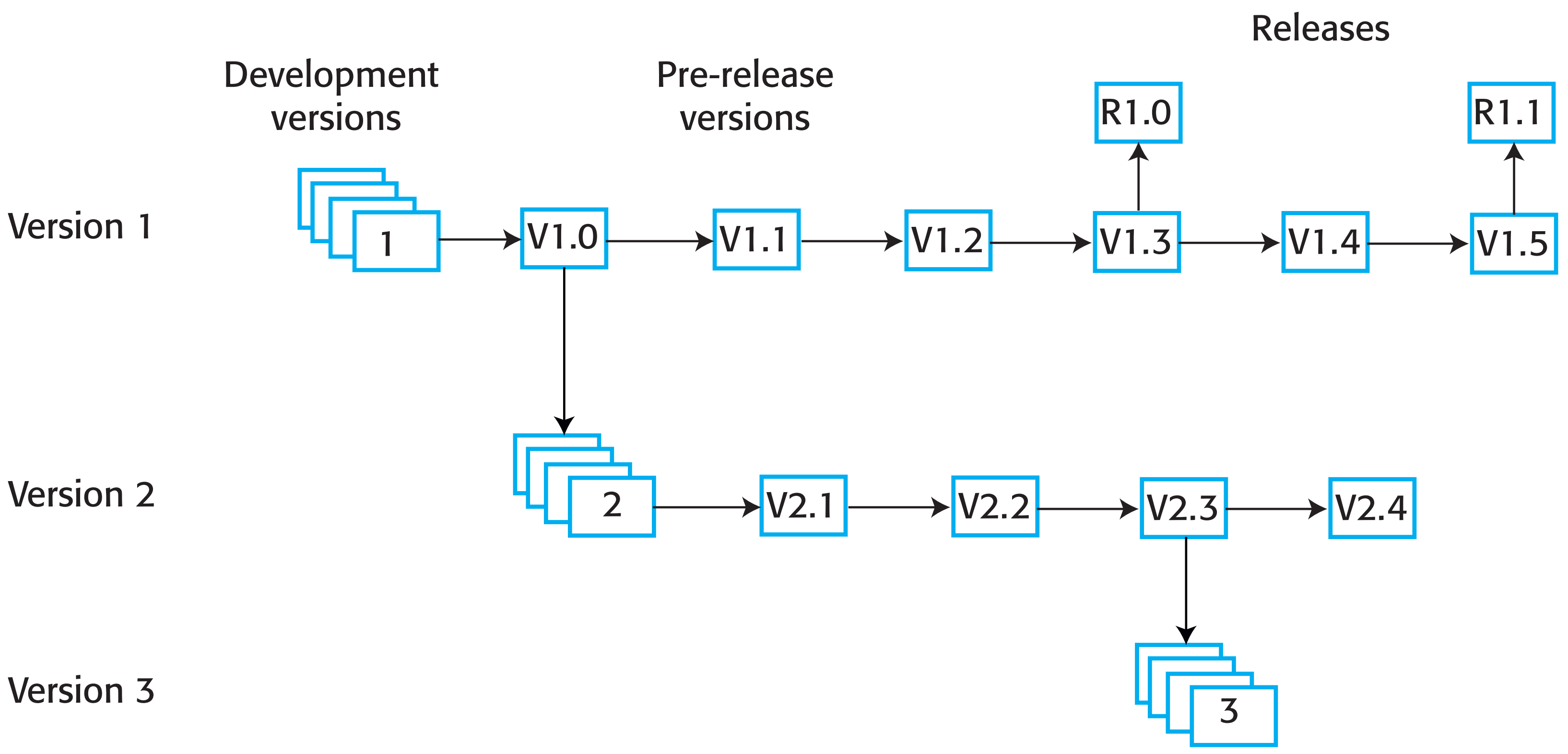
Development Phases

- A **development phase** where the development team is responsible for managing the software configuration and new functionality is being added to the software.
- A **system testing phase** where a version of the system is released internally for testing.
  - No new system functionality is added. Changes made are bug fixes, performance improvements and security vulnerability repairs.
- A **release phase** where the software is released to customers for use.
  - New versions of the released system are developed to repair bugs and vulnerabilities and to include new features.

Multi-version Systems

- For large systems, there is never just one ‘working’ version of a system.
- There are always several versions of the system at different stages of development.
- There may be several teams involved in the development of different system versions.

Multi-version System Development



CM Terminology

Term	Explanation
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, i.e., the versions of the components making up the system cannot be changed. It is always possible to recreate a baseline from its constituent components.
Branching	The creation of a new code-line from a version in an existing code-line. Both may then develop independently.
Code-line	A code-line is a set of versions of a software component and other configurations on which component depends.
Configuration (version) 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.
Configuration item	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.
Mainline	A sequence of baselines representing different versions of a system.
Merging	The creation of a new version of a software component by merging separate versions in different code-lines. These code-lines may have been created by a previous branch of one of the code-lines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
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.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

Version Management

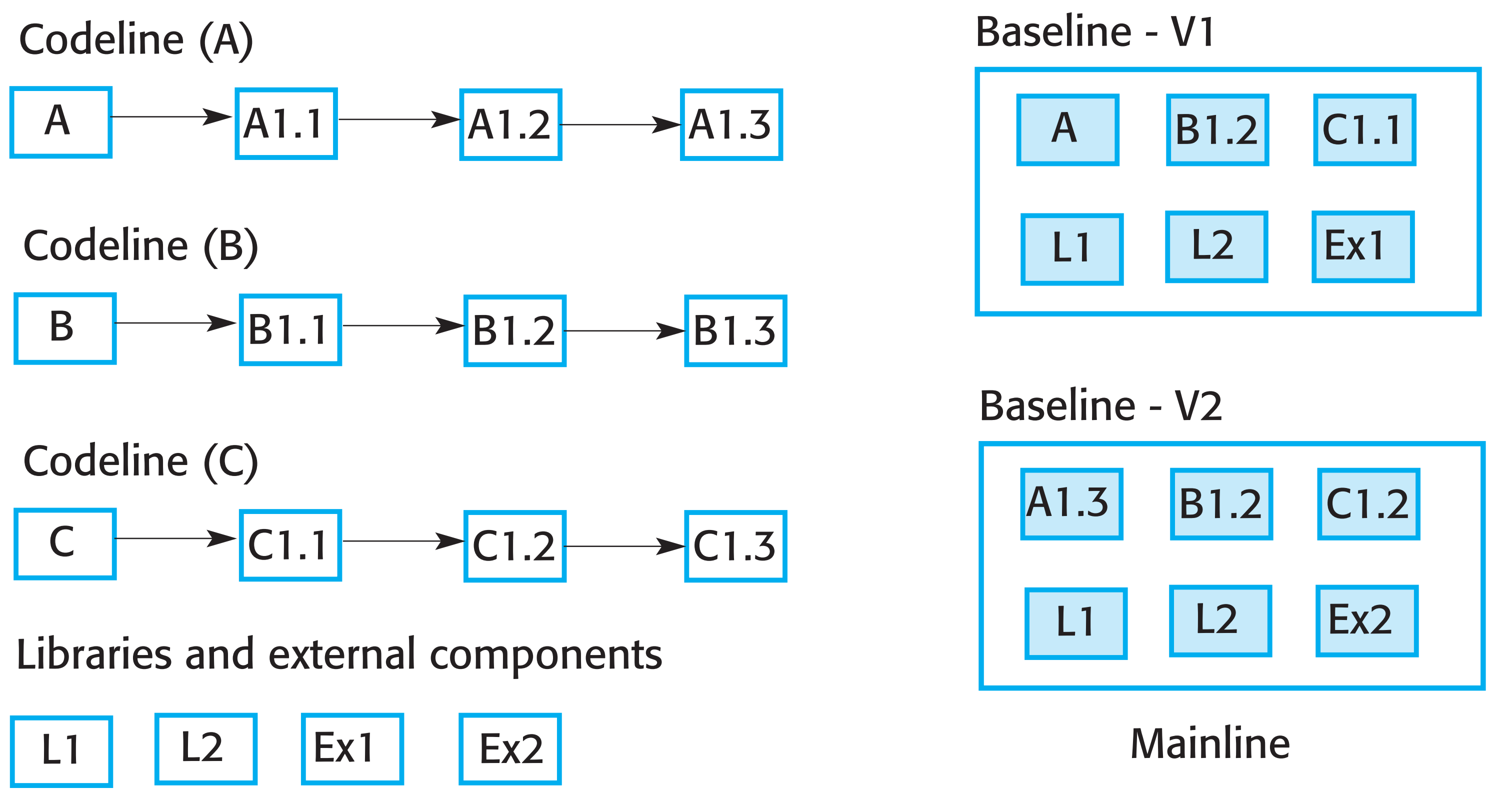
Version Management

- **Version management** (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- 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 Control Systems

- **Version control** (VC) systems identify, store and control access to the different versions of components. There are two types of modern version control system
  - **Centralized systems**, where there is a single master repository that maintains all versions of the software components that are being developed. ***Subversion*** is a widely used example of a centralized VC system.
  - **Distributed systems**, where multiple versions of the component repository exist at the same time. ***Git*** is a widely-used example of a distributed VC system.

Key Features of Version Control Systems

- Version and release identification
- Change history recording
- Support for independent development
- Project support
- Storage management

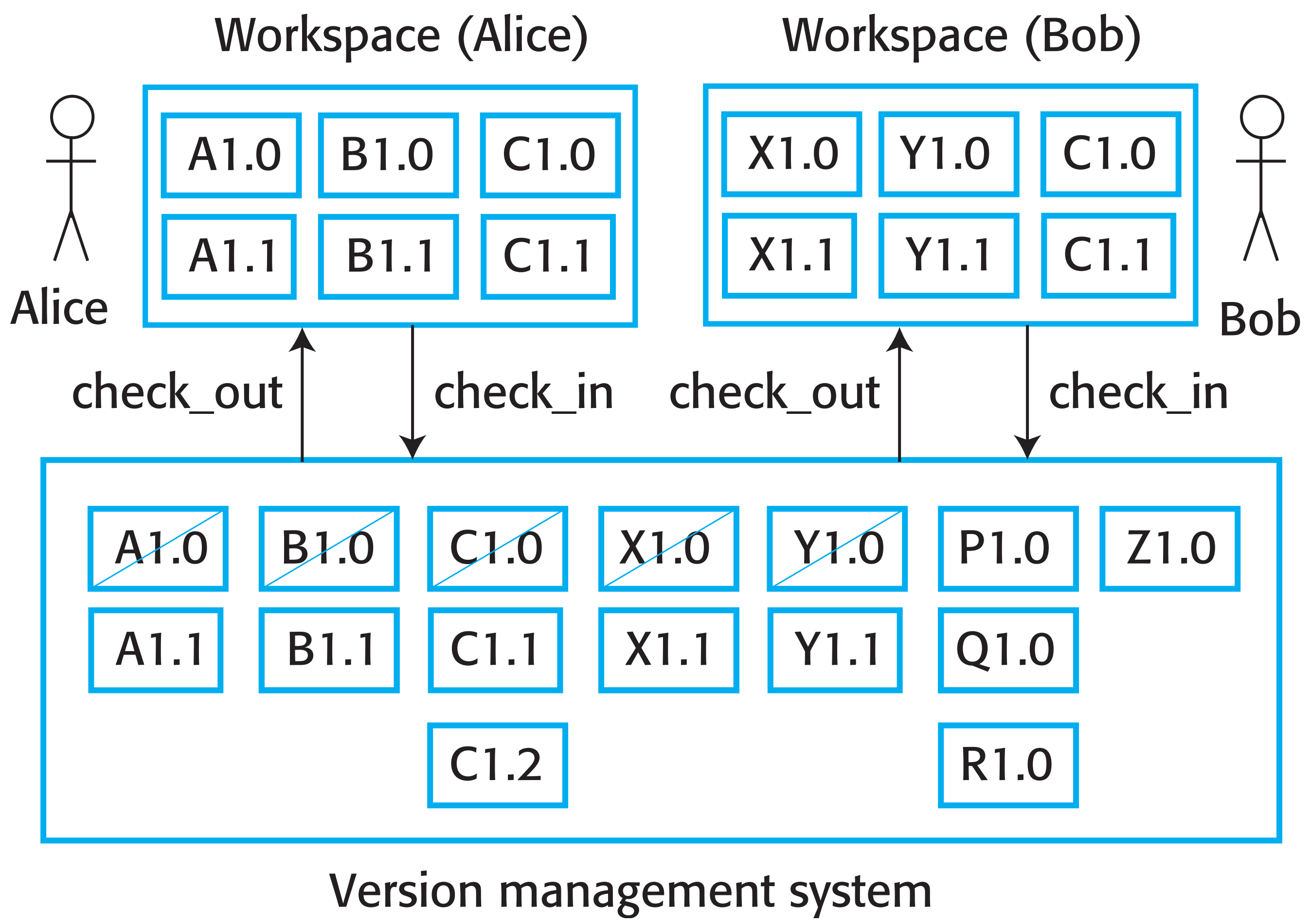
Public Repository and Private Workspaces

- To support independent development without interference, version control systems use the concept of a project repository and a private workspace.
- The project repository maintains the ‘master’ version of all components. It is used to create baselines for system building.
- When modifying components, developers copy (check-out) these from the repository into their workspace and work on these copies.
- When they have finished their changes, the changed components are returned (checked-in) to the repository.

Centralized Version Control

- Developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
- When their changes are complete, they check-in the components back to the repository.
- If several people are working on a component at the same time, each check it out from the repository. If a component has been checked out, the VC system warns other users wanting to check out that component that it has been checked out by someone else.

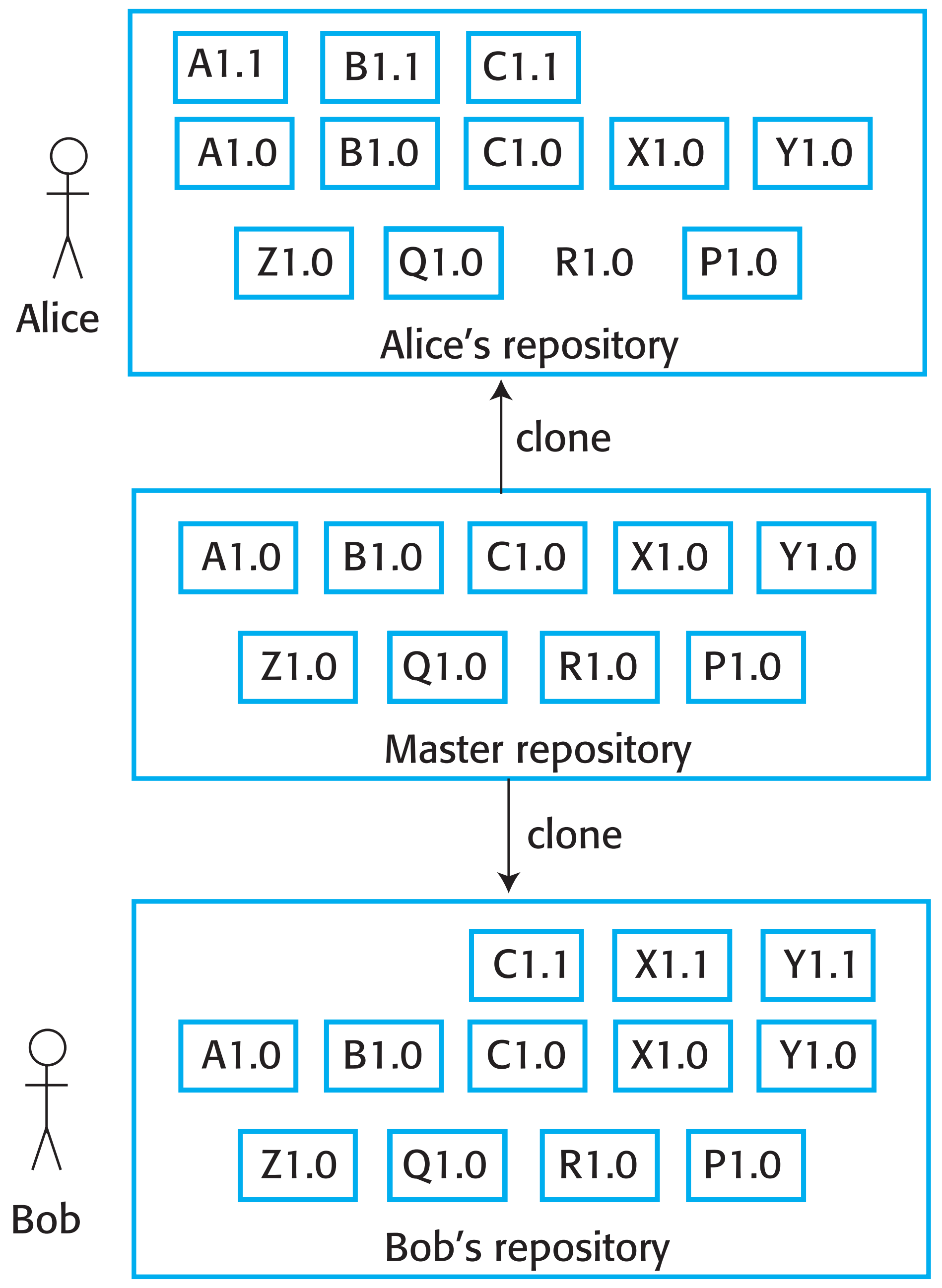
Repository Check-in/Check-out



Distributed Version Control

- A ‘master’ repository is created on a server that maintains the code produced by the development team.
- Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.
- Developers work on the files required and maintain the new versions on their private repository on their own computer.
- When changes are done, they ‘commit’ these changes and update their private server repository. They may then ‘push’ these changes to the project repository.

Repository Cloning



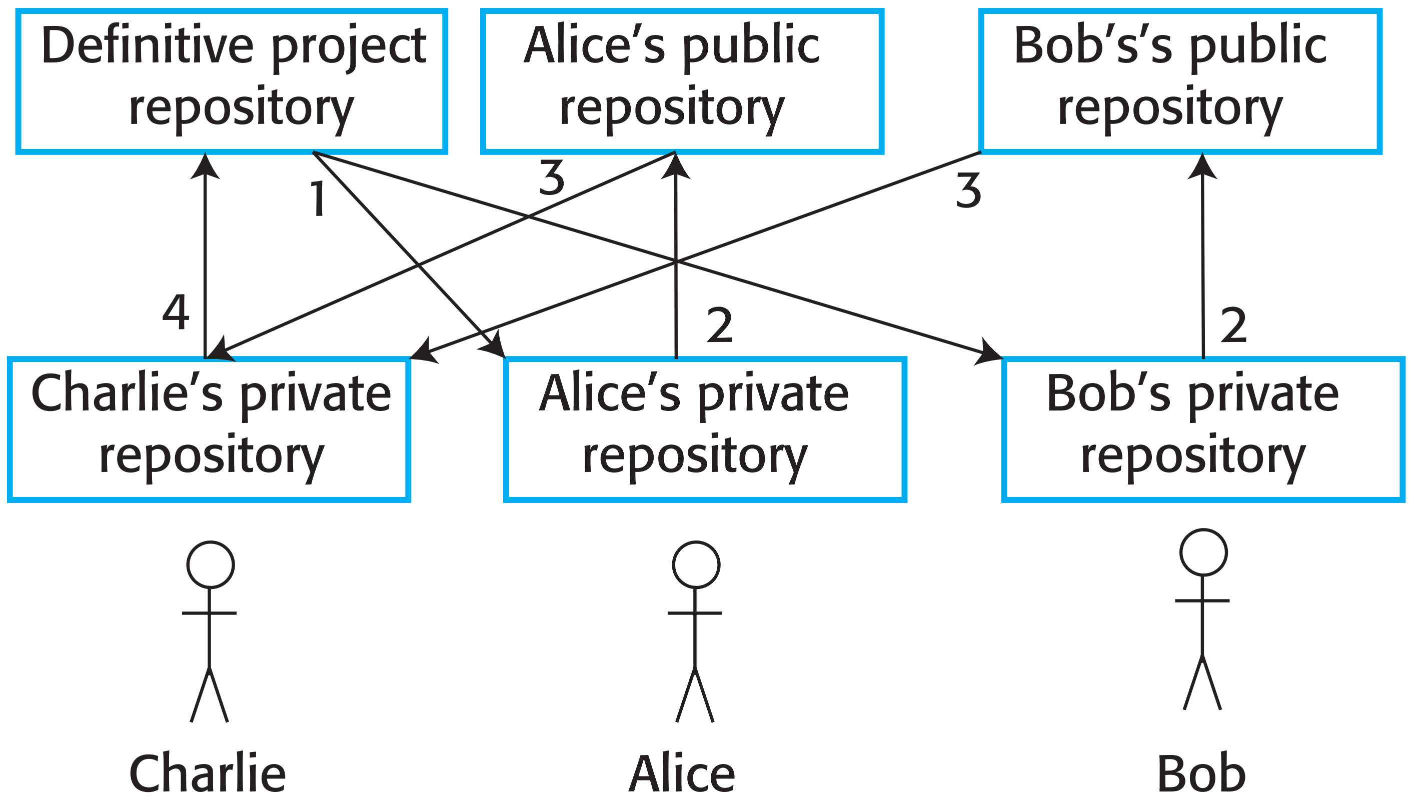
Benefits of Distributed Version Control

- It provides a **backup mechanism** for the repository.
    - If the repository is corrupted, work can continue and the project repository can be restored from local copies.
- It allows for **off-line working** so that developers can commit changes if they do not have a network connection.
- Project support is the default way of working.
    - Developers can compile and test the entire system on their local machines and test the changes that they have made.

Open Source Development

- Distributed version control is essential for open source development.
  - Several people may be working simultaneously on the same system without any central coordination.
- As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
  - It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

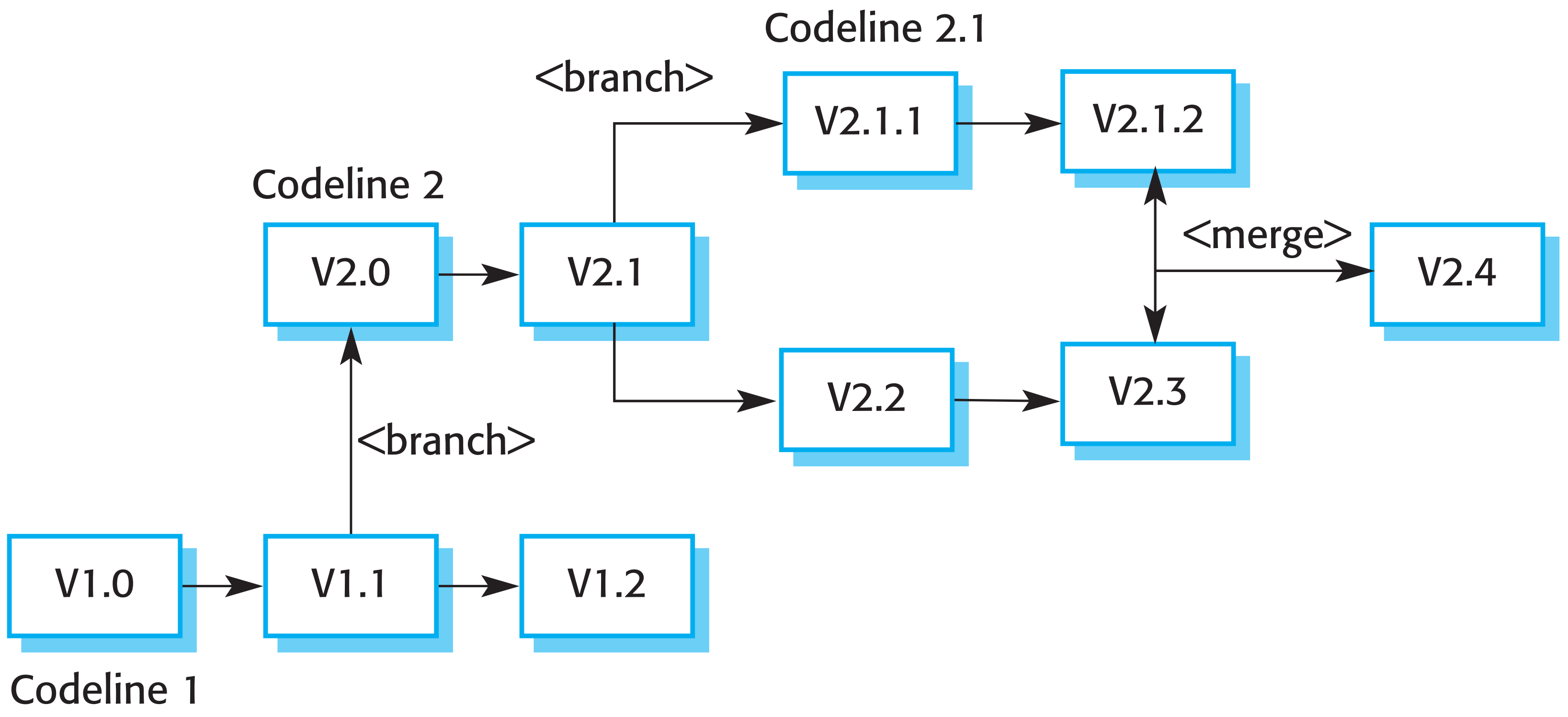
Open Source Development


Definitive project repository
Alice's public repository
Bob's's public repository
1
3
3
4
2
2
Charlie's private repository
Alice's private repository
Bob's private repository
Charlie
Alice
Bob

Branching and Merging

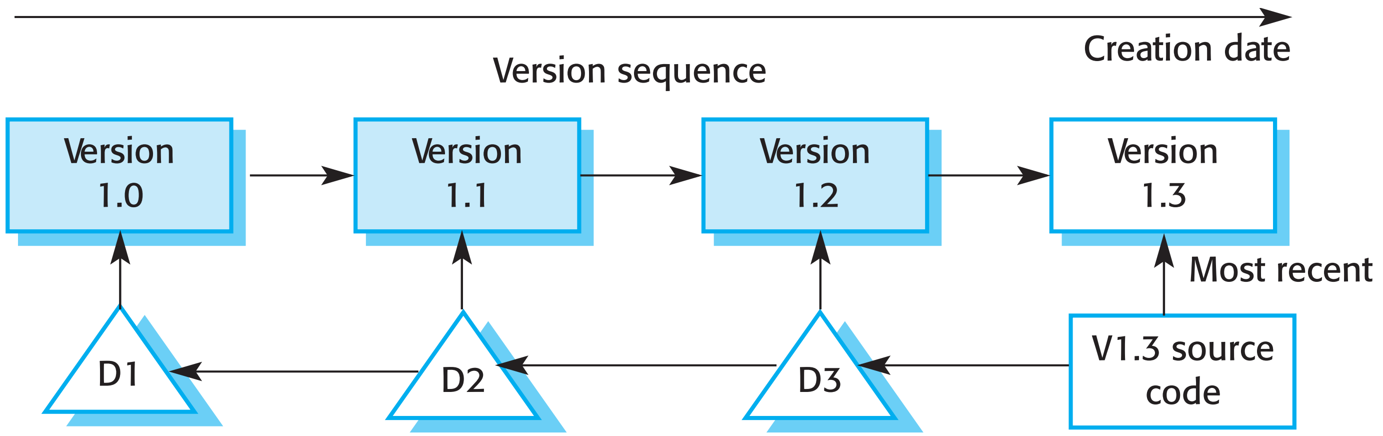
- Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.
  - This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- 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



Storage Management

- When version control systems were first developed, storage management was one of their most important functions.
- Disk space was expensive and it was important to minimize the disk space used by the different copies of components.
- Instead of keeping a complete copy of each version, the system stores **a list of differences (deltas) between one version and another**.
    - By applying these to a master version (usually the most recent version), a target version can be recreated.



Storage Management in Git

- As disk storage is now relatively cheap, Git uses an alternative, faster approach.
- Git does not use deltas but applies a **standard compression algorithm to stored files and their associated meta-information**.
- It does not store duplicate copies of files. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations.
- Git also uses the notion of packfiles where several smaller files are combined into an indexed single file.

System Building

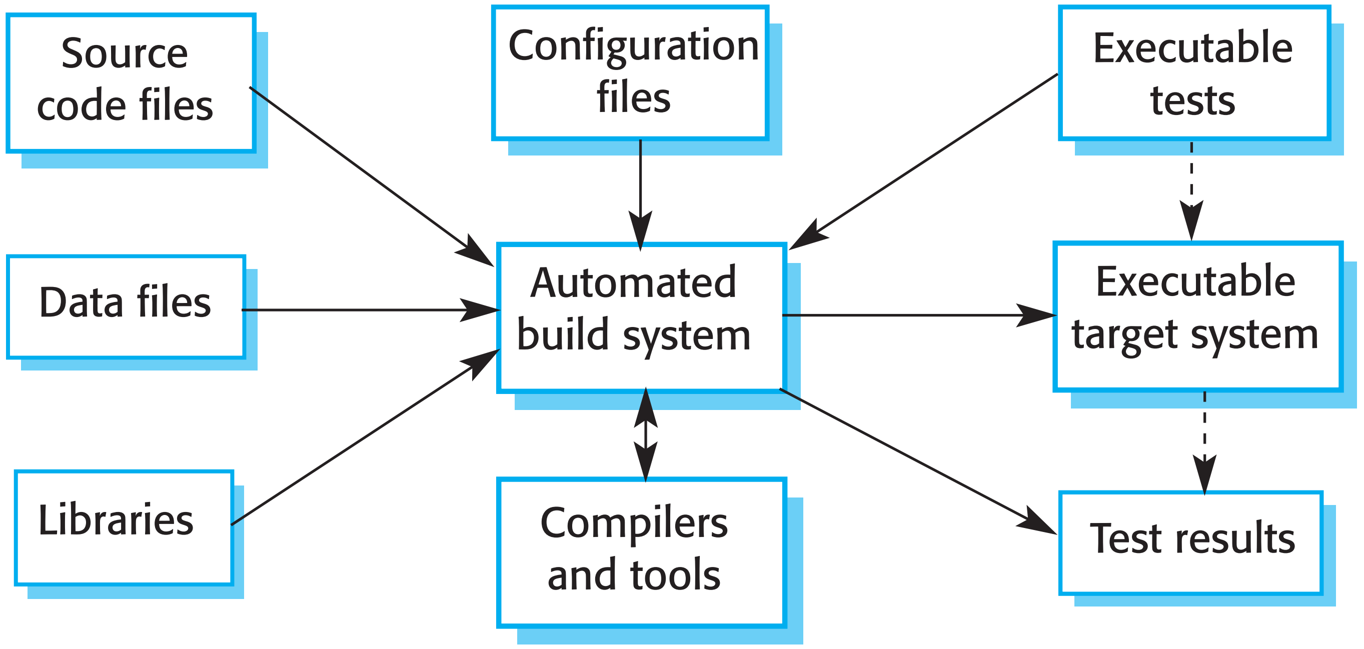
System Building

- **System building** is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- The **configuration descri**ption used to identify a baseline is also used by the system building tool.

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.

System Building



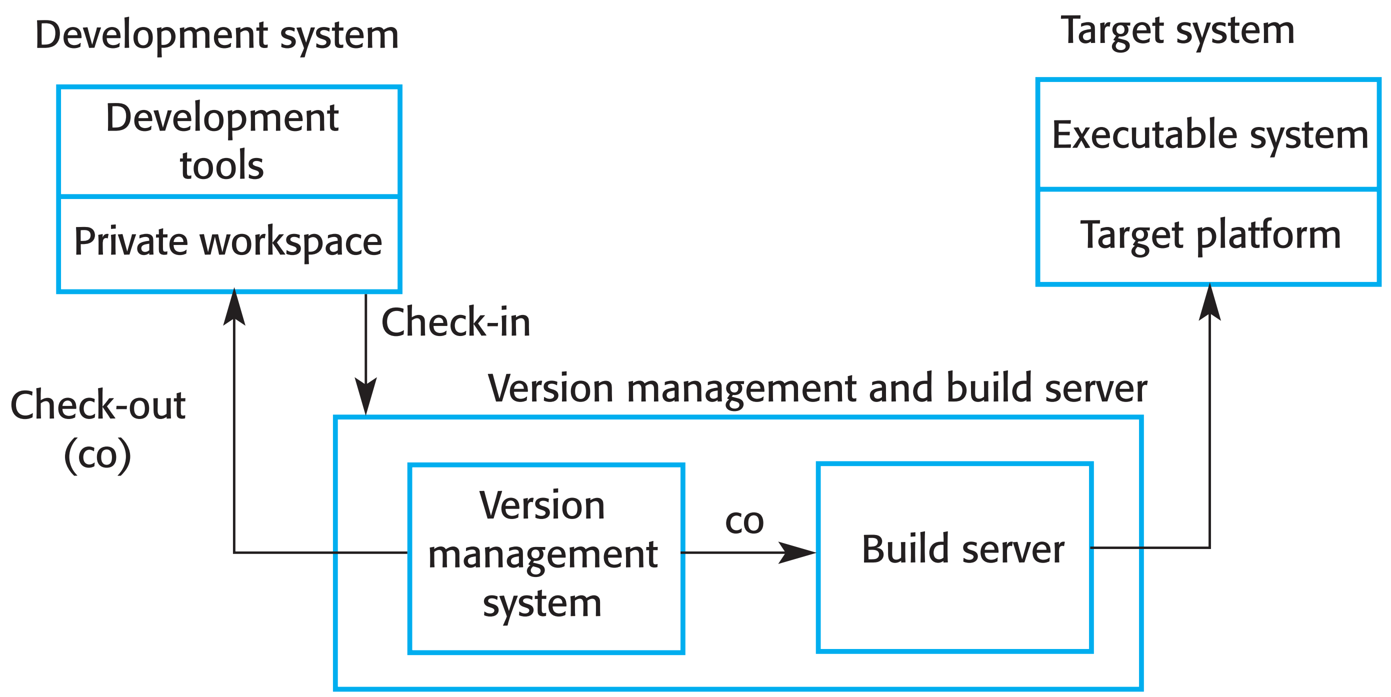
Build System Functionality

- Build script generation
- Version management system integration
- Minimal re-compilation
- Executable system creation
- Test automation
- Reporting
- Documentation generation

System Platforms

- The development system, which includes development tools such as compilers, source code editors, etc.
- The build server, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system.
- The target environment, which is the platform on which the system executes.
  - For real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g. a cell phone)

Development, Build, and Target Platforms



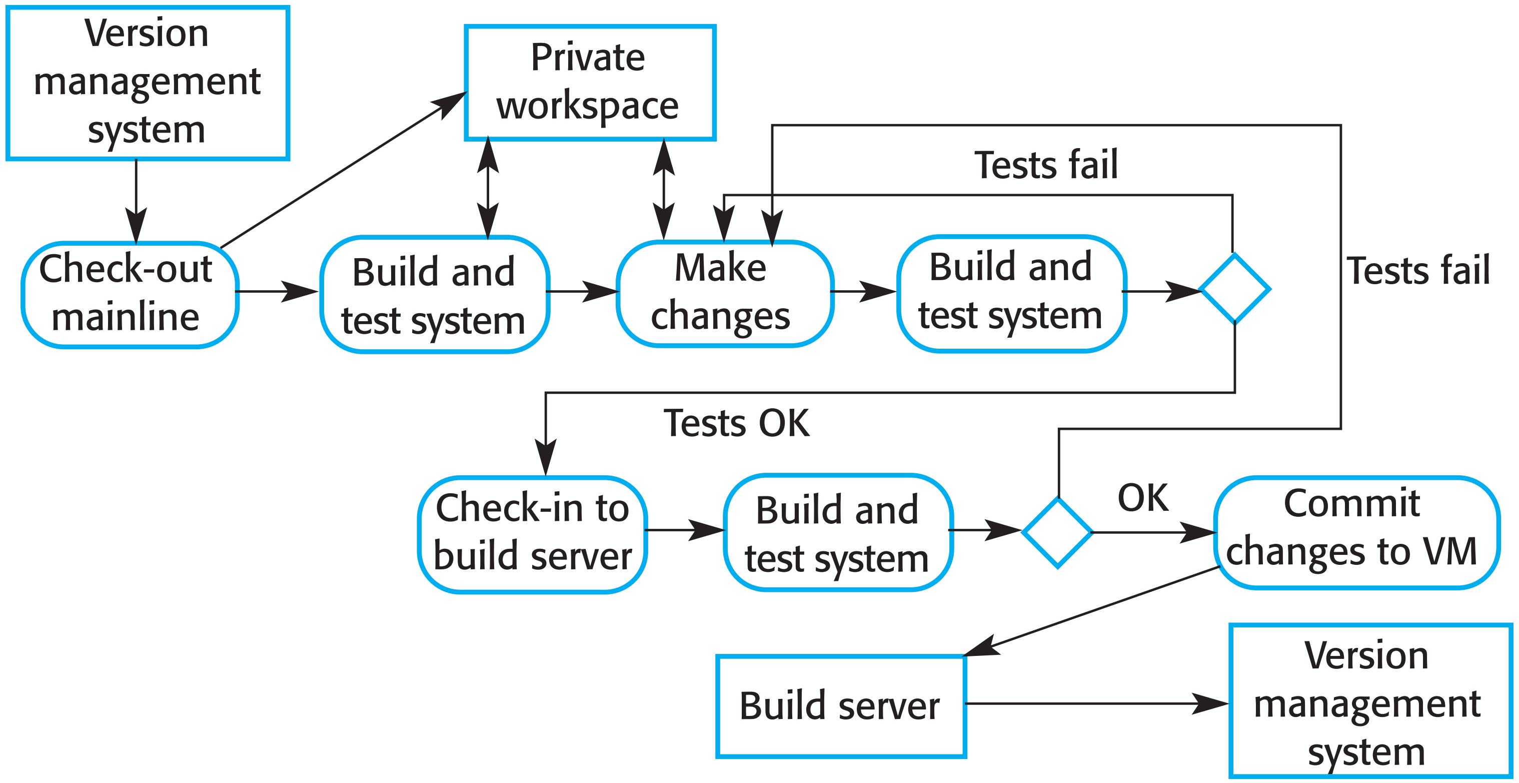
Agile Building

- Check out the mainline system from the version management system into the developer's private workspace.
- Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
- Make the changes to the system components.
- Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.

Agile Building

- Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.
- Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
- If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

Continuous Integration



Pros and Cons of Continuous Integration

- Pros
  - The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible.
  - The most recent system in the mainline is the definitive working system.
- Cons
  - If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved.
  - If the development platform is different from the target platform, it may not be possible to run system tests in the developer’s private workspace.

Daily Building

- The development organization sets a delivery time (say 2 p.m.) for system components.
  - If developers have new versions of the components that they are writing, they must deliver them by that time.
  - A new version of the system is built from these components by compiling and linking them to form a complete system.
  - This system is then delivered to the testing team, which carries out a set of predefined system tests
  - Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

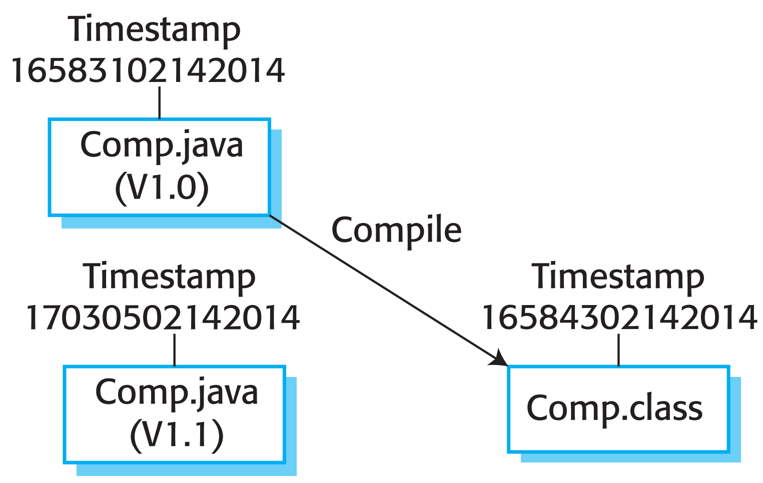
Minimizing Recompilation

- Tools to support system building are usually designed to **minimize the amount of compilation that is required**.
- They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- A unique signature identifies each source and object code version and is changed when the source code is edited.
- By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.

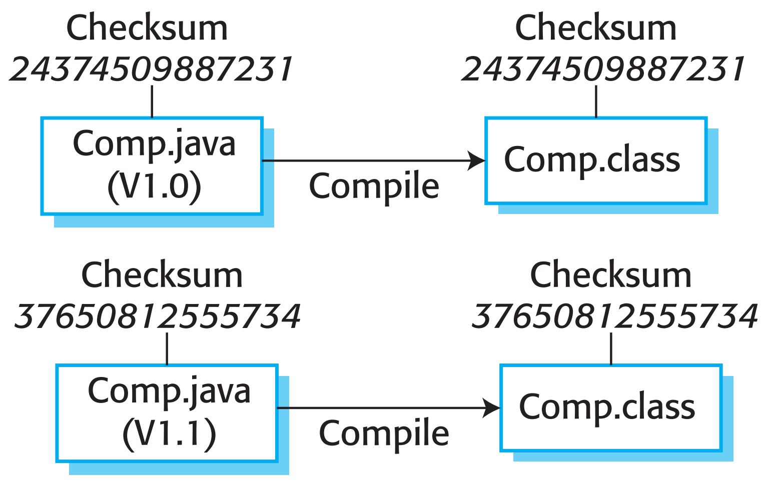
File Identification

- **Modification timestamps**
  - The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.
- **Source code checksums**
  - The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.

Linking Source and Object Code



Time-based identification



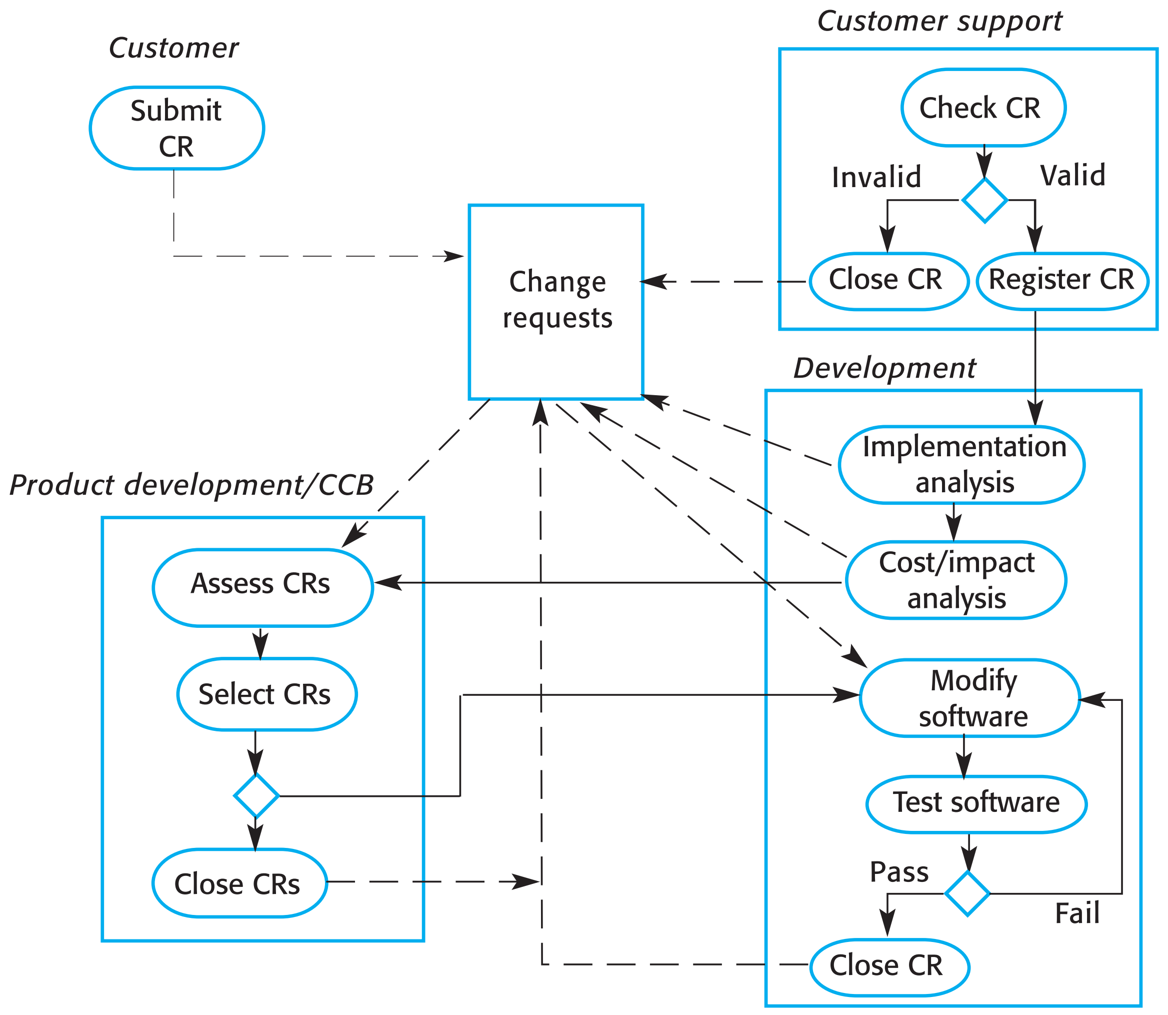
Checksum-based identification

Change Management

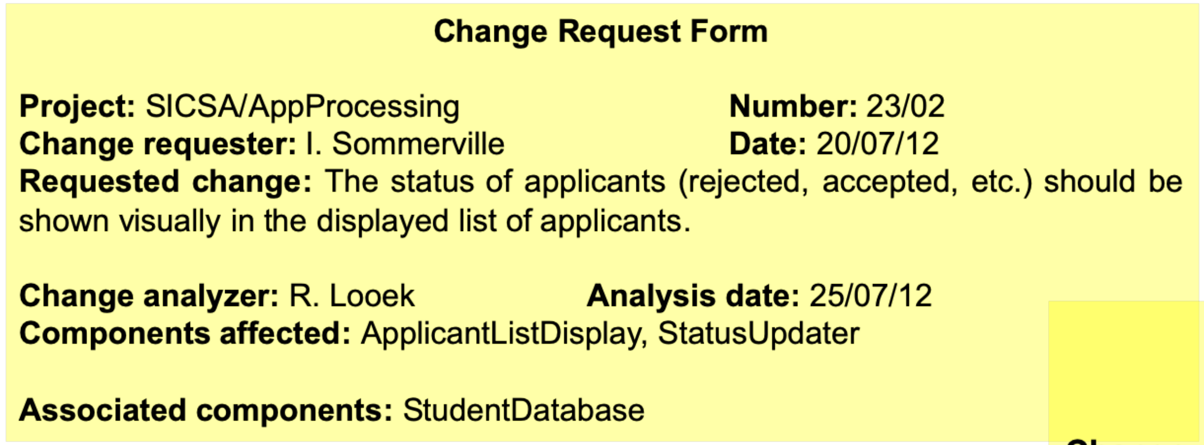
Change Management

- Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- 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.
- The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.

Change Management Process



Partially Completed Change Request Form

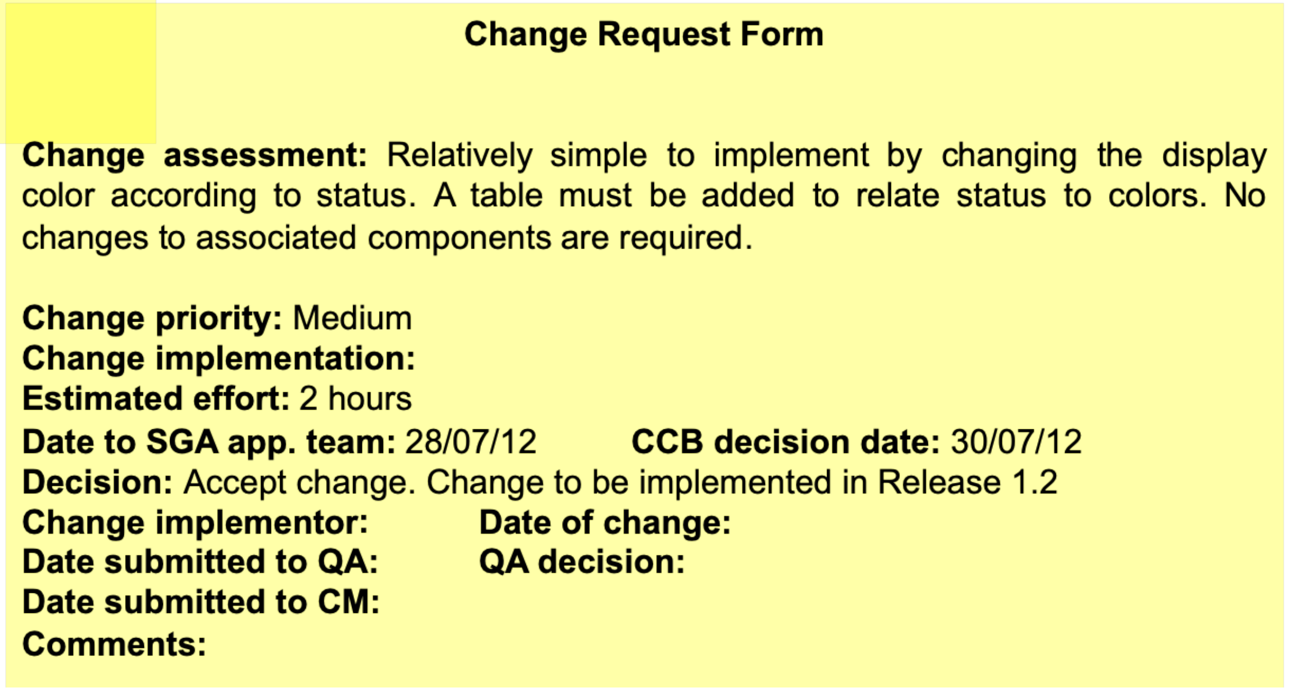


Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/07/12
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek **Analysis date:** 25/07/12
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase



Change Request Form

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium
Change implementation:
Estimated effort: 2 hours
Date to SGA app. team: 28/07/12 **CCB decision date:** 30/07/12
Decision: Accept change. Change to be implemented in Release 1.2
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments:

Change Management and Agile Methods

- In some agile methods, customers are directly involved in change management.
- The propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- Changes to improve the software improvement are decided by the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

Release Management

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.
- For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

Release Components

- As well as the the executable code of the system, 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.

Factors Influencing System Release Planning

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.

Release Creation

- The executable code of the programs and all associated data files must be identified in the version control system.
- Configuration descriptions may have to be written for different hardware and operating systems.
- Update instructions may have to be written for customers who need to configure their own systems.
- Scripts for the installation program may have to be written.
- Web pages have to be created describing the release, with links to system documentation.
- When all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

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.
- This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
    - Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

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 Planning

- As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- Release timing
  - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
  - If system releases are too infrequent, market share may be lost as customers move to alternative systems.

Software as a Service

- Delivering software as a service (SaaS) reduces the problems of release management.
- It simplifies both release management and system installation for customers.
- The software developer is responsible for replacing the existing release of a system with a new release and this is made available to all customers at the same time.

つづく