COMP 3111
SOFTWARE ENGINEERING

LECTURE 12
IMPLEMENTATION

WAYS TO GET YOUR CODE RIGHT

- **Defensive programming**
  - Programming with testing and debugging in mind.

- **Testing**
  - For uncovering problems and increasing confidence (next topic).

- **Debugging**
  - Finding out why a program is not functioning as intended.

- **Testing ≠ Debugging**

  Testing: Reveals the existence of a problem.

  Debugging: Pinpoints the location *plus* the cause of a problem.

DEFENDING AGAINST BUGS

1. Make errors impossible by design

2. Do not introduce defects

3. Make errors immediately visible

4. Last resort is debugging

DEBUGGING DEFENSE I:
MAKE ERRORS IMPOSSIBLE BY DESIGN

- **In the language**
  - E.g., Java makes memory overwrite bugs impossible.

- **In the protocols/libraries/modules**
  - TCP/IP will guarantee that data is not reordered.
  - Java BigInteger will guarantee that there will be no overflow.

- **In self-imposed conventions**
  - Hierarchical locking makes deadlock bugs impossible.
  - Banning the use of recursion will make infinite recursion/insufficient stack bugs go away.
  - Immutable data structures will guarantee behavioural equality.

☞ **Caution: You must maintain the discipline!**

DEBUGGING DEFENSE 2:
DO NOT INTRODUCE DEFECTS

- **Get things right the first time**
  - **Do not code before you think! Think before you code!**
  - If you are making lots of easy-to-find bugs, you are also making hard-to-find bugs. → ***Do not use the compiler as a crutch.***
- **Especially true when debugging is going to be hard**
  - Concurrency
  - Difficult test and instrument environments
  - Program must meet timing deadlines
- **Simplicity is the key**
  - **Modularity**
    - Divide program into chunks that are easy to understand
    - Use abstract data types with well-defined interfaces
    - Use defensive programming
  - **Specification**
    - Write specifications for all modules so that an explicit, well-defined interface exists for each module that clients can rely on.

DEBUGGING DEFENSE 3:
MAKE ERRORS IMMEDIATELY VISIBLE

General Approach: fail-fast!

☞ **Try to localize bugs to a small part of the program.**

- **Take advantage of modularity**
  - Start with everything and take away pieces until bug disappears.
  - Start with nothing and add pieces back in until bug appears.

- **Take advantage of modular reasoning**
  - Trace through program, viewing intermediate results.

- **Use binary search to speed things up**
  - Bug happens somewhere between first and last statement.
  - Do a binary search on that ordered set of statements.

☞ **Make use of assertions where possible.**

EXAMPLE: OBSCURING A BUG

```
// k is guaranteed to be present in a
int i = 0;
while (true)
{
  if (a[i] == k)
  break;
  i++;
}
```

This code fragment searches an array a for a value k.

The value is guaranteed to be in the array.

If that guarantee is broken (by a bug), the code throws an exception and dies.

Temptation: Make the code more “robust” by not failing.

EXAMPLE: OBSCURING A BUG (cont'd)

```
// k is guaranteed to be present in a
int i = 0;
while (i < a.length)
{
  if (a[i] == k)
  break;
  i++;
}
```

Now at least the loop will always terminate, BUT …

EXAMPLE: OBSCURING A BUG (cont'd)

```
// k is guaranteed to be present in a
int i = 0;
while (i < a.length)
{
  if (a[i] == k)
  break;
  i++;
}
assert (i < a.length): "Value not found in a[].";
```

Assertions let us document and check invariants.

☞ **Abort the program as soon as a problem is detected.**

WHERE IS THE BUG?

- **The bug is not where you think it is.**
  - Ask yourself where it cannot be; explain why.

- **Look for stupid mistakes first, such as:**
  - Reversed order of arguments: Collection.copy(src, dest)
  - (Mis)Spelling of identifiers
  - Same object versus equals: a == b versus a.equals(b)
  - Failure to reinitialize a variable
  - Deep versus shallow copy

- **Make sure that you have the correct source code.**
  - Recompile everything

INSERTING CHECKS

- **Insert lots of checks with an intelligent checking strategy.**
  - Precondition checks, consistency checks, bug-specific checks
- **Goal: Stop the program as close to a bug as possible.**
  - Use debugger to see where you are, explore the program a bit.
- **Should assertions/checks be included in production code?**

  Yes Stop the program if check fails—don't want to take a chance that the program will do something wrong.

  No May need the program to keep going, maybe bug does not have such bad consequences.

  ☞ **The correct answer depends on the context!**

REGRESSION TESTING

- **Whenever you find and fix a bug**
  - Add a test for it
  - Re-run all your tests

- **Why is this a good idea?**

- **Run regression tests as frequently as you can afford to.**
  - Automate the process.
  - Make concise test sets with few unnecessary tests.

DEBUGGING: THE LAST RESORT

- **Bugs happen.**

  Industry average: **10** bugs per 1000 lines of code.

- **Bugs that are not immediately localized happen.**
  - Found during integration testing.
  - Reported by user.

Step 1: Clarify the symptom.

Step 2: Find and understand the cause; create a test.

Step 3: Fix.

Step 4: Rerun all tests.

DEBUGGING: WHEN THE GOING GETS TOUGH

- **Reconsider assumptions**
  - E.g., has the OS changed? Is there room on the hard drive?
  - Debug the code, not the comments.
- **Start documenting your system**
  - Gives a fresh angle and highlights areas of confusion.
- **Get help**
  - We all develop blind spots.
  - Explaining the problem to others often helps.
- **Walk away**
  - Trade latency for efficiency—**sleep**!
  - One good reason to start early.

CONFIGURATION MANAGEMENT (CM)

***Configuration management* manages, controls and monitors changes to life cycle artifacts.**

- **Change management**
  - Tracks and evaluates proposed developer/client software changes.
- **Version management**
  - Tracks and manages multiple versions of system components.
- **System building**
  - Creates an executable system from components, data and libraries.
- **Release management**
  - Prepares and tracks system versions released to customers.

CHANGE MANAGEMENT

Change management **ensures that system evolution is a managed process.**

☞ **To give priority to the most urgent and cost-effective changes.**

To what do we want to control changes?

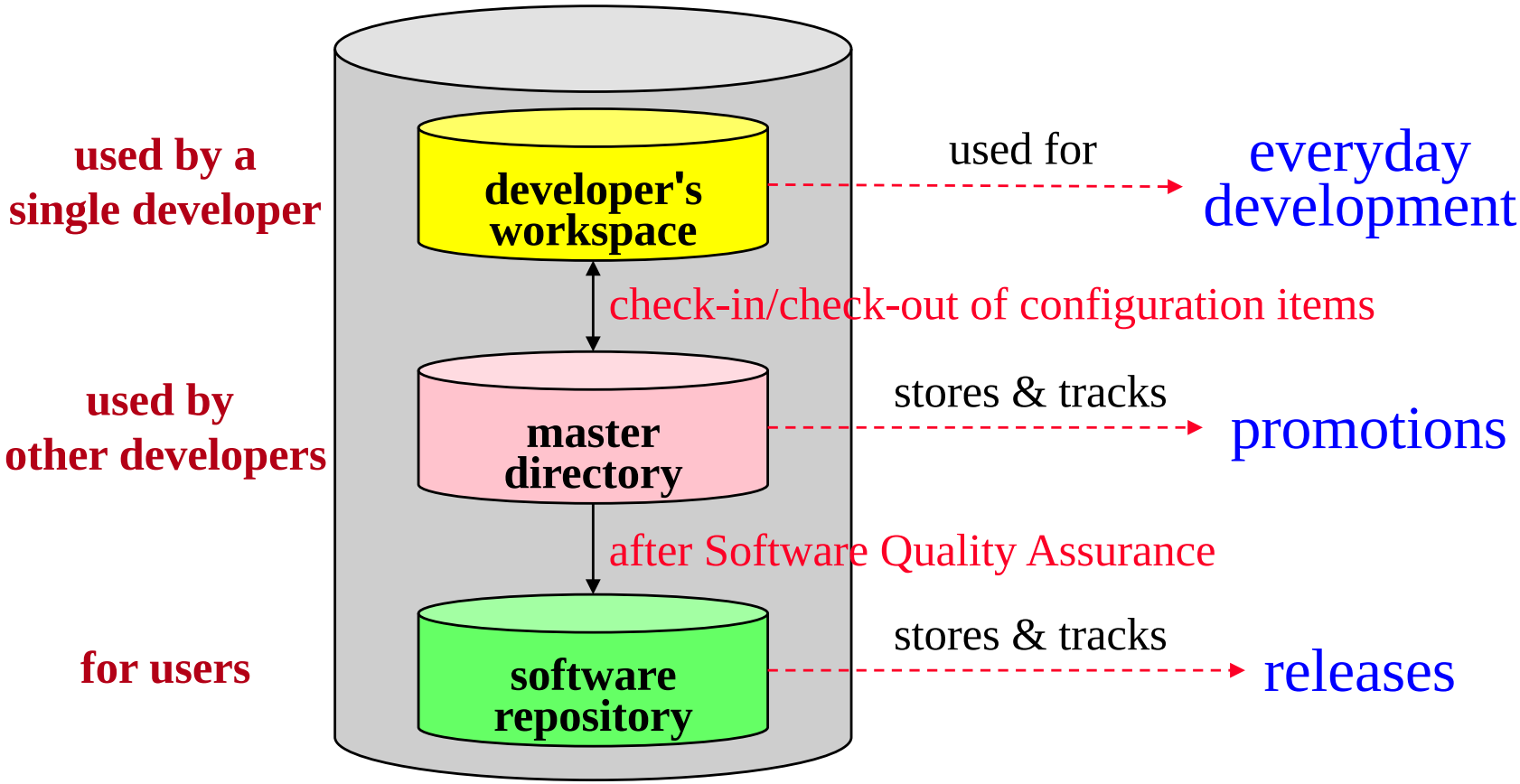
- plans
- specifications
- procedures
- programs/code
- documents/manuals
- data

identify — support — control — audit — report

configuration item: an artifact to which we want to control changes

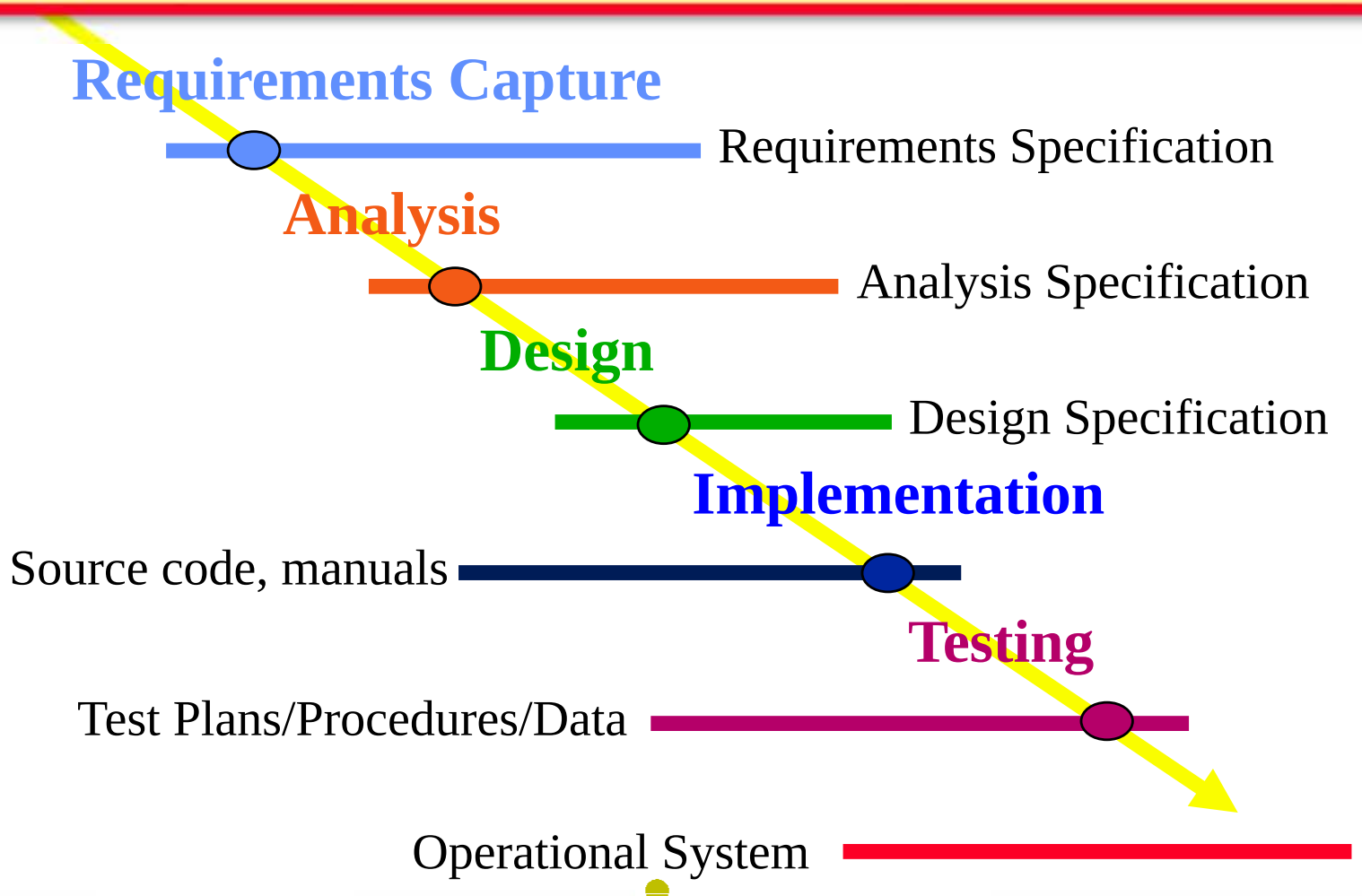
CHANGE MANAGEMENT: SUPPORT

A *software library* provides facilities to store, label and identify versions and to track the status of the configuration items.



CHANGE MANAGEMENT: CHANGE CONTROL

A *baseline* is a time/phase in the software development after which any changes must be formalized (i.e., be controlled).



CHANGE MANAGEMENT: CHANGE CONTROL

☞ **A baseline defines a specific system.**
(component versions, libraries, configuration files, etc.)

- To become part of a baseline, a configuration item must first pass a set of formal review procedures (e.g., documentation review, code review, etc.) → usually at a project milestone.

- It then becomes part of the project software library.

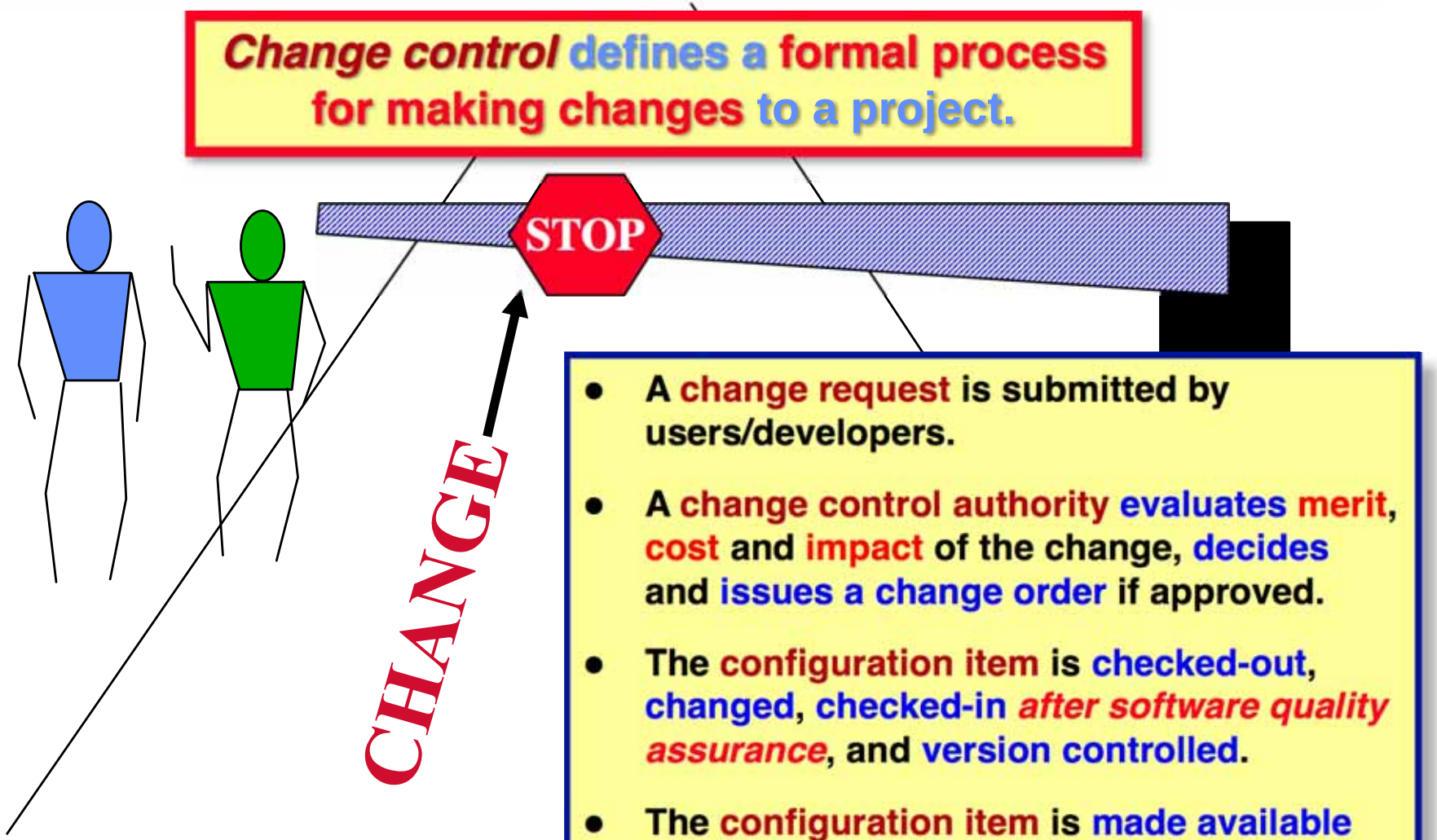
☞ **After this a "check-out" procedure is applied to the item.**
(i.e., access to and change of the configuration item is controlled)

- Any modified configuration item must again go through a formal review process before it can replace the original (baseline) item.

☞ **Requires version management.**

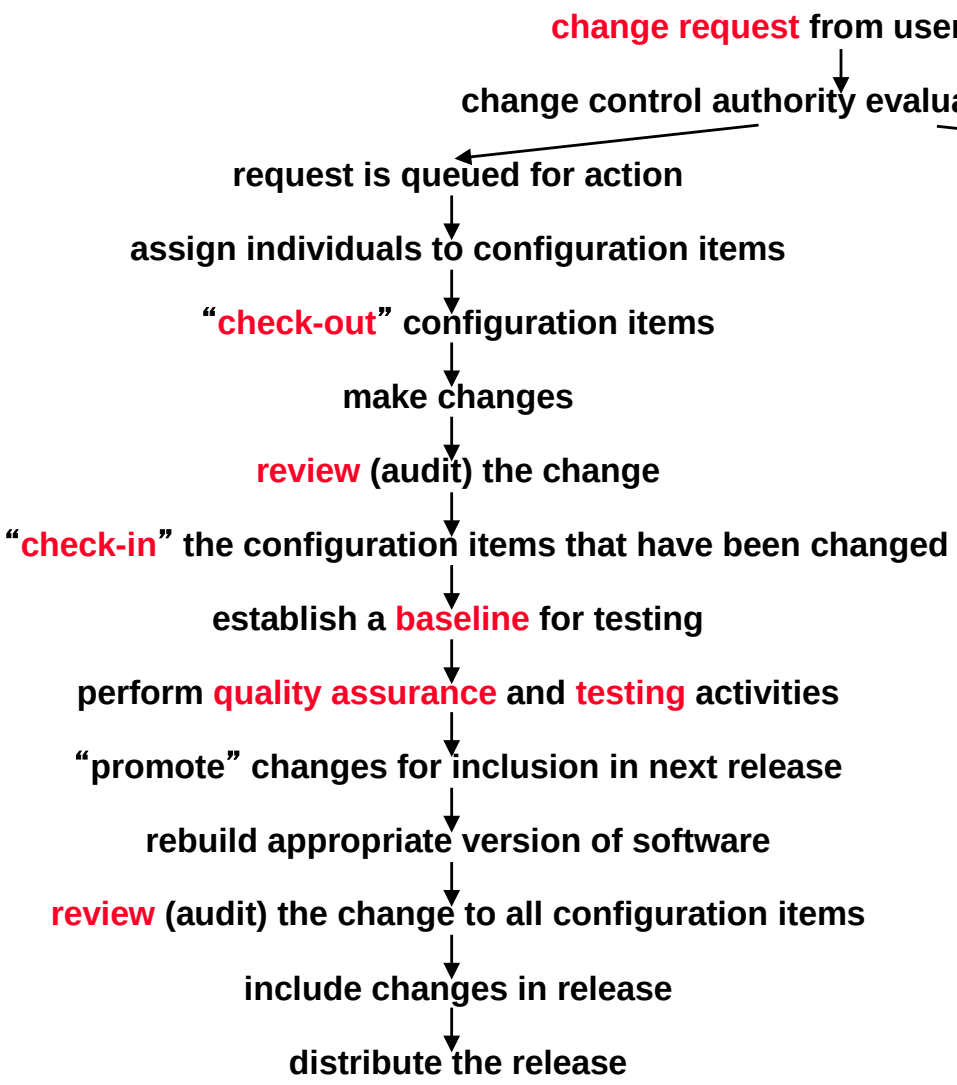
CHANGE MANAGEMENT: CHANGE CONTROL

Change control **defines a formal process for making changes to a project.**



- A **change request** is submitted by users/developers.
- A **change control authority** evaluates **merit**, **cost** and **impact** of the change, **decides** and **issues a change order** if approved.
- The **configuration item** is **checked-out**, **changed**, **checked-in** ***after software quality assurance***, and **version controlled**.
- The **configuration item** is **made available for use** (**promotion**; **release**).

CHANGE MANAGEMENT: CHANGE CONTROL



Can be applied to
internal requests
(i.e., user = developer)
and
external requests
(i.e., user = end-user).

CHANGE MANAGEMENT: AUDITING AND STATUS REPORTING

Auditing: ensures that changes have been properly implemented.

- We need to verify that the proper steps and procedures have been followed when making a change.
- This is usually done by a Quality Assurance (QA) group if SCM is a formal activity in the organization.

Status reporting: keeps all parties informed and up-to-date on the status of a change.

- Status reporting is a communication mechanism among project members to help keep them coordinated.
- Status reporting allows management/users to determine who made what changes, when and why.

VERSION MANAGEMENT

Version management **ensures the integrity and consistency of configuration items.**

☞ **By managing different versions of an item.**

codeline a sequence of versions of source code with later versions derived from earlier versions.

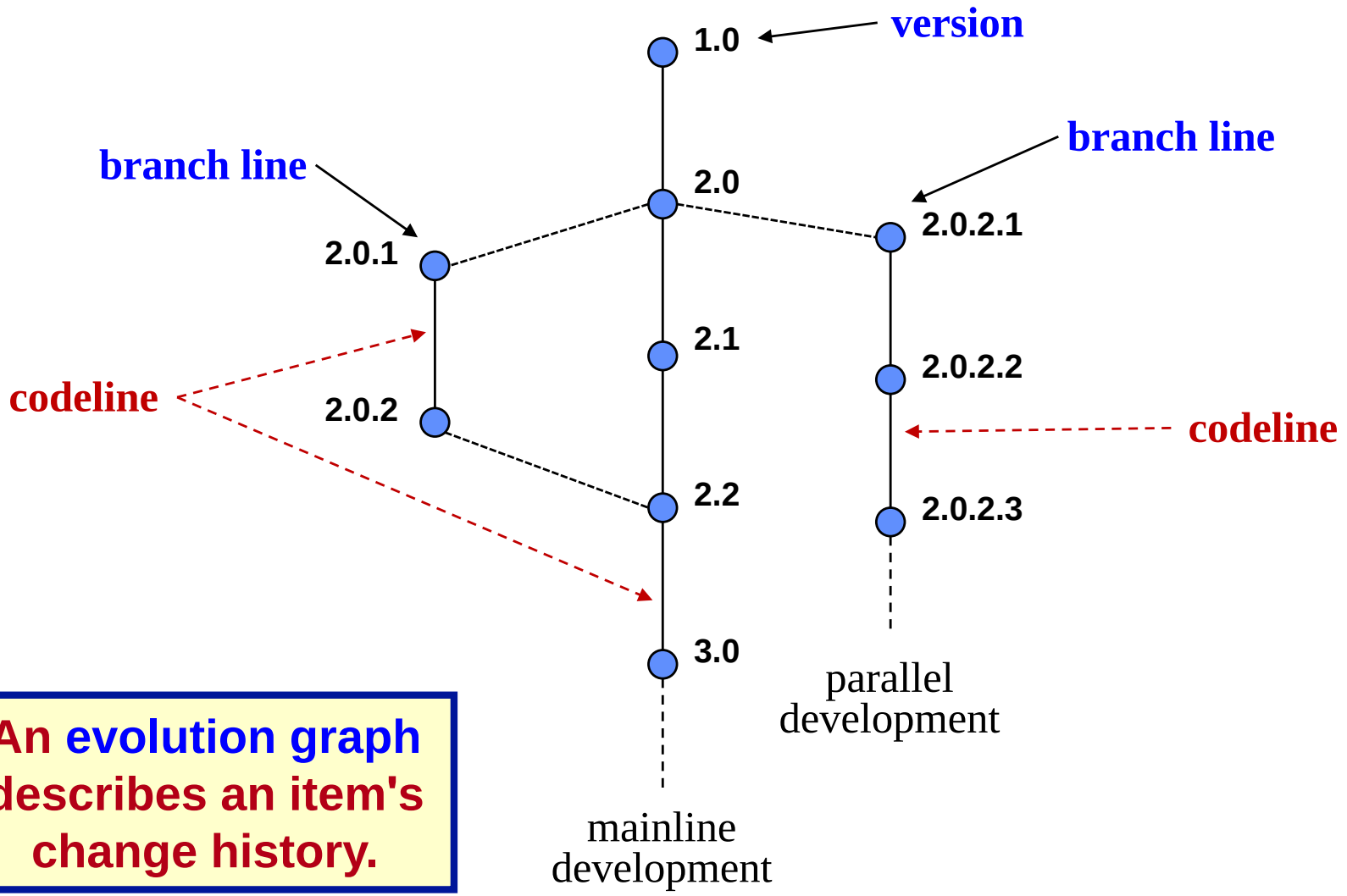
version configuration $item_k$ is obtained by modifying configuration $item_i$ and supersedes it; the items are created in a linear order.

branch A concurrent development path requiring independent configuration management.

variant Different configurations that are intended to coexist.

E.g., Oracle for Windows Oracle for Linux

VERSION MANAGEMENT



An evolution graph describes an item's change history.

IMPLEMENTATION: SUMMARY

- **The Implementation workflow:**
  - Implements classes (primarily operation methods) in modules.
  - Organizes modules into subsystems.
  - Integrates all modules and subsystems into the final system.
  - Assigns executable modules to processing nodes.
- **Producing solid code requires skills in:**
  - Defensive programming
  - Code review
  - Refactoring
- **Debugging**
  - Removes faults in the code
- **Configuration Management**
  - Helps ensure product quality by controlling changes