Static Verification & Validation Techniques
Software Reviews and Inspections

Daniel Varró ♦ McGill University
daniel.varro@mcgill.ca

Based on: ISTQB Foundational Syllabus, R. Patton ch.6, D. Galin ch.8
Other sources: L. Bass, P. Clements, R. Kazman

Overview

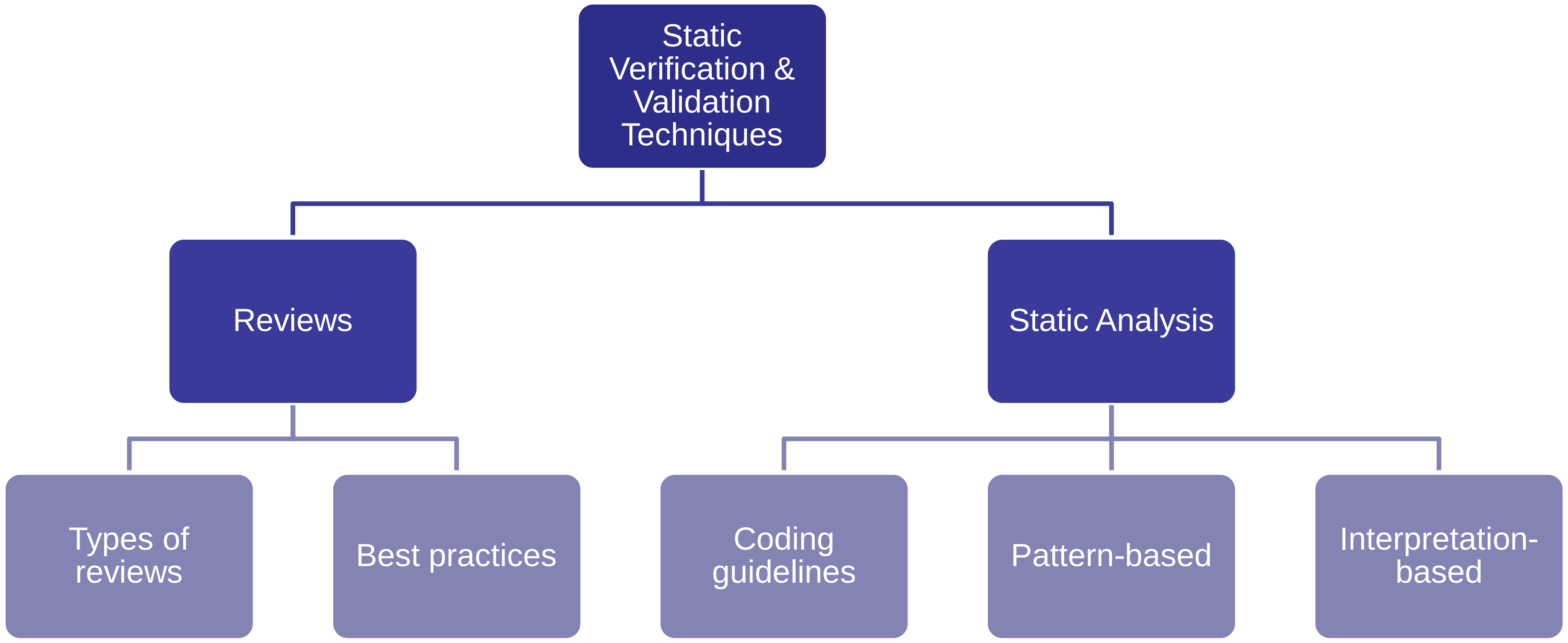
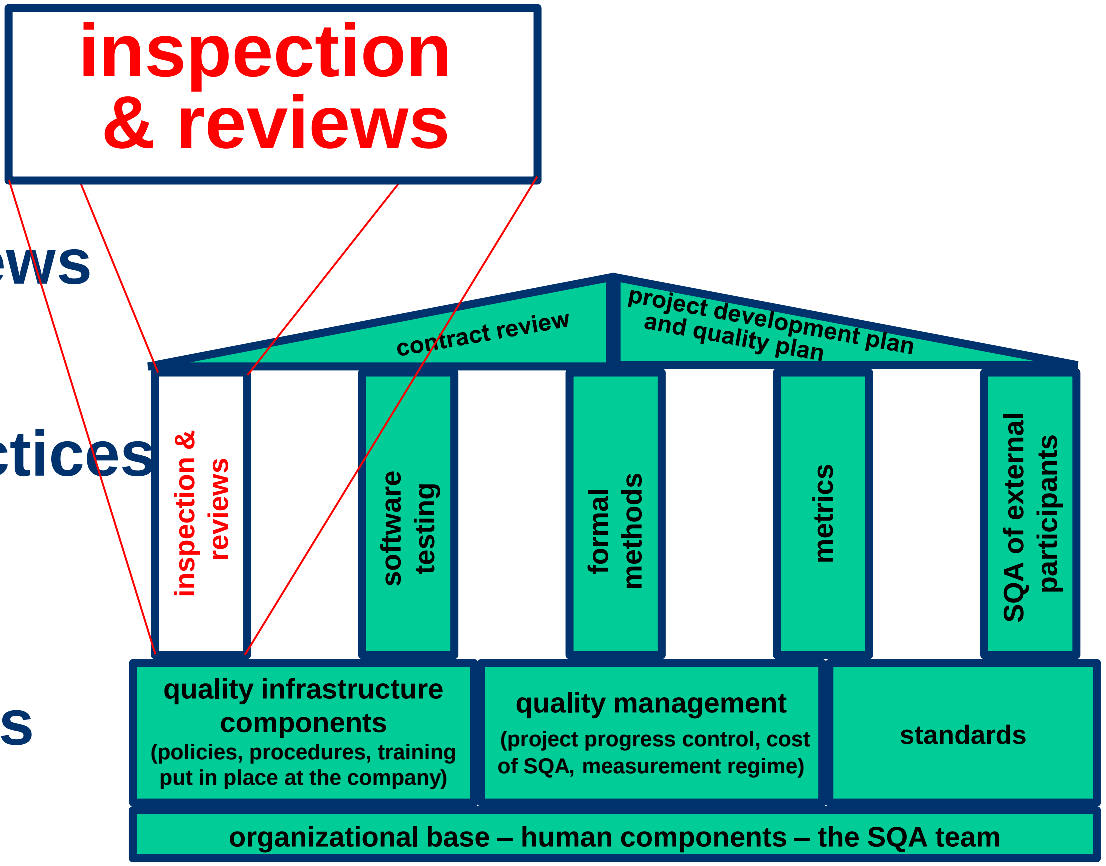

Static Verification & Validation Techniques
Reviews
Static Analysis
Types of reviews
Best practices
Coding guidelines
Pattern-based
Interpretation-based

Table of Contents



inspection & reviews
contract review
project development plan and quality plan
inspection & reviews
software testing
formal methods
metrics
SQA of external participants
quality infrastructure components
(policies, procedures, training put in place at the company)
quality management
(project progress control, cost of SQA, measurement regime)
standards
organizational base – human components – the SQA team

Sec. 3.1 in ISTQB Foundational Level Syllabus

STATIC V&V TECHNIQUES

Scope of Static V&V

- **No execution of work product**
  - Manual examination → reviews
  - Tool-driven evaluation → static analysis
- **Target application domains:**
  - Traditionally: safety-critical systems
  - Nowadays: any software with CI / automated build
- **Target work products:**
  - Any work product that can be read and understood
    - Specifications, Requirements, User stories, etc.
    - Architecture, Design
    - Models
    - Code
    - Testware
    - Documentation, User guides

Benefits of Static V&V

- **Enables early detection of defects**
  - Before dynamic V&V
  - No executability needed
- **Cheaper to remove defects when found early**
- **Main benefits:**
  - Detect and correct defects prior to dynamic testing
  - Identify defects not found by dynamic testing
  - Prevent defects in design or coding
  - Increase productivity (more maintainable code)
  - Reduce cost and time of development and testing
  - Improve communication between team members

Static vs. Dynamic Testing

- **Both have similar objectives**
  - Assess quality, identify defects as early as possible, ...
- **Dynamic testing**
  - Requires the execution of the software
    → identify failures caused by defects
  - Improve externally visible behavior
- **Static testing**
  - Finds defects directly in work products
    → a defect may not cause a failure for long time
  - Improves consistency and internal quality of WP
- **Can complement each other**

Effectiveness of Static V&V

- **Typical defects found effectively by static V&V**
  - **Requirement defects**: e.g., inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies
  - Design defects: e.g., inefficient algorithms or database structures, high coupling, low cohesion
  - **Coding defects**: e.g., variables with undefined values, variables declared but never used, unreachable code, duplicate code
  - Deviations from coding standards
  - **Incorrect interface specifications**: e.g., different units of measurement used by the caller than by the called system
  - **Security vulnerabilities**: e.g., susceptibility to buffer overflows
  - Gaps or inaccuracies in test basis traceability or coverage: e.g., missing tests for an acceptance criterion
  - **Maintainability defects**

Sec. 3.2 in ISTQB Foundational Level Syllabus

REVIEW PROCESS

Review Process

- **IEEE, 1990**

review process = a process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval

- **Purposes:**
  - Identify defects (errors or deviations from templates/style) and new risks
  - informally exchange knowledge
  - collect data to learn from mistakes
- **Human examination of any work products**
  - Requirements, design, source code, test cases, etc.

Types of Reviews

Informal Review

- Informal, Performed by development team (e.g. agile)
- **Main purpose**: detect potential defects

Walkthrough

- Mostly informal, Driven by author of work product
- **Main purposes**: find defects, improve the software product, consider alternative implementations, evaluate conformance to standards and specifications

Technical Review

- Documented process, Experts are involved
- **Main purposes**: gain consensus, detect potential defects

Inspection

- Formally documented process, external experts, moderators involved
- **Main purposes**: detect potential defects, evaluate quality and build confidence in the work product, prevent future similar defects through author learning and root cause analysis

Increasing level of formality

http://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html

A formal review process

INSPECTIONS

Inspections

- **Strict, very formal form of review**
- **To obtain defects, collect data, communicate development documents**
- **Team of 3 to 6 persons**
- **Roles:**
  - Moderator
  - Recorder (scribe)
  - Reviewer
  - Reader (presenter)
  - Producer (author)

Checklists

- **Most important tool for formal reviews**
- **Generic checklists for various types of reviews:**
  - Requirements
  - Design
  - Generic code
  - Specific language code (C, C++, Java, etc.)
  - Generic document
- **Organizations develop specific checklists:**
  - Particular objectives, past experience, etc.
- **Should be maintained, improved, developed, updated**

(see Appendix B for more details)

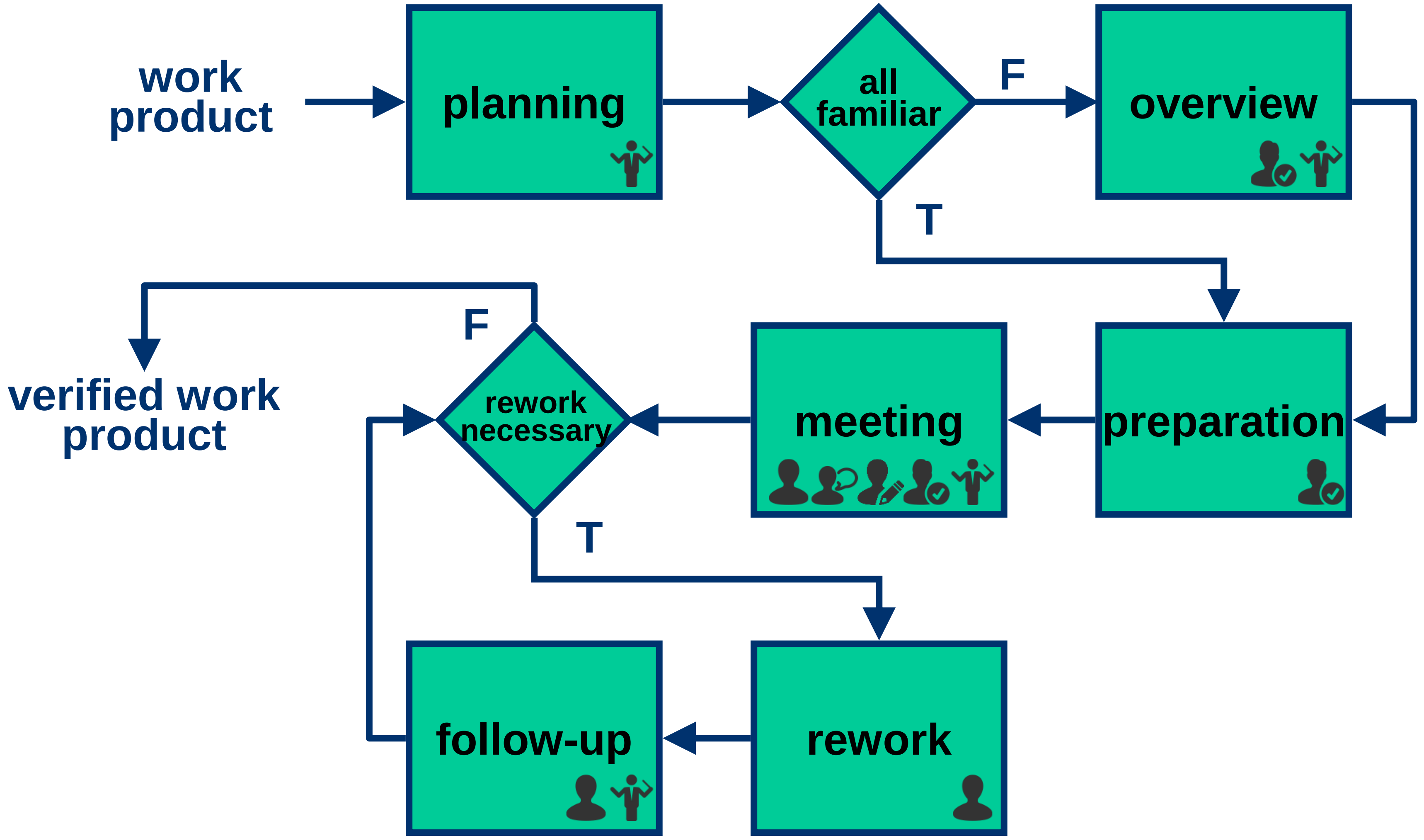
Inspections: Roles

- **Moderator:**
  - Crucial role for an inspection
  - Ensures that the inspection procedures are followed
  - Verifies the work product's readiness for inspection
  - Verifies that the entry criteria are met
  - Assembles an effective inspection team
  - Keeps the inspection meeting on track
  - Verifies that the exit criteria are met
  - Comes from outside the project team (but still a peer)
- **Recorder:**
  - Documents all defects that arise from the inspection meeting in an inspection defect list
  - Not just a procedural task – requires technical knowledge

Inspections: Roles

- **Reviewer:**
  - Analyzes and detects defects in the work product
  - All participants play that role
  - Depending on reviewed document, consider a representative each from requirements (or user), test, design, and implementation teams
- **Reader:**
  - Leads the inspection team through the inspection meeting by reading aloud small logical units, paraphrasing where appropriate
- **Producer:**
  - Work product author
  - Responsible for correcting any found defects

Inspection Process



… moderator … reviewer … recorder … presenter … producer

icons from: http://www.visualpharm.com/

Inspection Process

- **Planning**:
  - Identify work product, determine whether the product to be inspected meets the entry criteria, select the team, assign roles, prepare and distribute the inspection forms and materials, set the inspection schedule, and determine whether to hold an overview

- **Overview**:
  - Optional phase where team members who are unfamiliar with the work product to be inspected receive orientation

Inspection Process

- **Preparation**:
  - Key stage in which members of the inspection team inspect the work individually looking for defects in the work product
  - Most of the defects are found during this step, not during the inspection meeting
- **Inspection meeting**:
  - Inspection team members meet to find, categorize, and record possible defects in the work product
  - No resolution of defects is attempted but action items are assigned
  - Beware of team size and long, detailed presentations

Inspection Process

- **Third hour:**
  - Optional additional time, apart from the inspection meeting, that can be used for discussion, possible solutions, or closure of open issues raised during the inspection meeting
  - If actual review task needs more than 2 hours because of the complexity involved, schedule another review session (not more than two a day)

Inspection Process

- **Rework**:
  - The work product is revised by the author to correct defects found during the inspection
- **Follow up**:
  - Meeting between the moderator and author to determine if defects found during the inspection meeting have been corrected and to ensure that no additional defects have been introduced
  - Final inspection data is collected and summarized, and the inspection is officially closed

Review Reports

- **Rate the severity of a defect**
- **Possibly determine statistics about findings and invested resources, quality/efficiency metrics**
- **For formal inspection/reviews, reports may be required:**
  - Inspection/FDR meeting notice report:
    - To launch review process
  - FDR report
    - Captures decision, summary, list of action items including anticipated completion date, person responsible for action item, and person responsible for checking completion of action item

(see Appendix B for more details)

Effectiveness of Defect Detection Methods at Fujitsu

Year	Defect Detection Method			Defects per 1000 lines of code*
	Test %	Design Review %	Code Inspection %	
1	85	---	15	0.19
2	80	5	15	0.13
3	70	10	20	0.06
4	60	15	25	0.05
5	40	30	30	0.04

*** defects in maintained code, detected by users within the first six months of regular software system use**

Inspections: Positive Side Effects

- Learn from each other
- Fosters team spirit (when well run)
- May help testing team identify problem areas/test cases to focus on
- Gives management an idea of whether the project is on track

Identify Review Types

Exercise!

Characterisitics	InfRev	Walkth	TecRev	Insp
Scribe is mandatory				
Review meeting is mandatory				
Individual preparation before review is mandatory for everyone				
Use of checklists is optional				
Defect logs and review reports produced				
Review meeting is led by the author				
Reviewers should be peers of author or technial experts in discipline				
Follows a formal, documented process				
Review meeting led by a facilitator				

Comparison of Techniques

inspection	walkthrough	informal review
presenter is not the author	anyone can be presenter	no presenter
three to six participants	two to seven participants	two or three participants
participants are peers	participants are peers	participants are peers
preparation essential	only presenter needs to prepare	preparation not required
follow-up on corrective actions	no follow-up on corrective actions	no follow-up on corrective actions
data is collected	data collection not required	data not collected
effective but short-term costs	less effort / finds fewer defects	inexpensive / finds fewer defects

Modern Code Reviews

Traditional vs. Modern Reviews

Mandated reviewer checklists and in-person meetings

Lightweight, tool-supported, flexible, asynchronous

Who uses modern reviews?

Modern Reviews in Open Source Systems

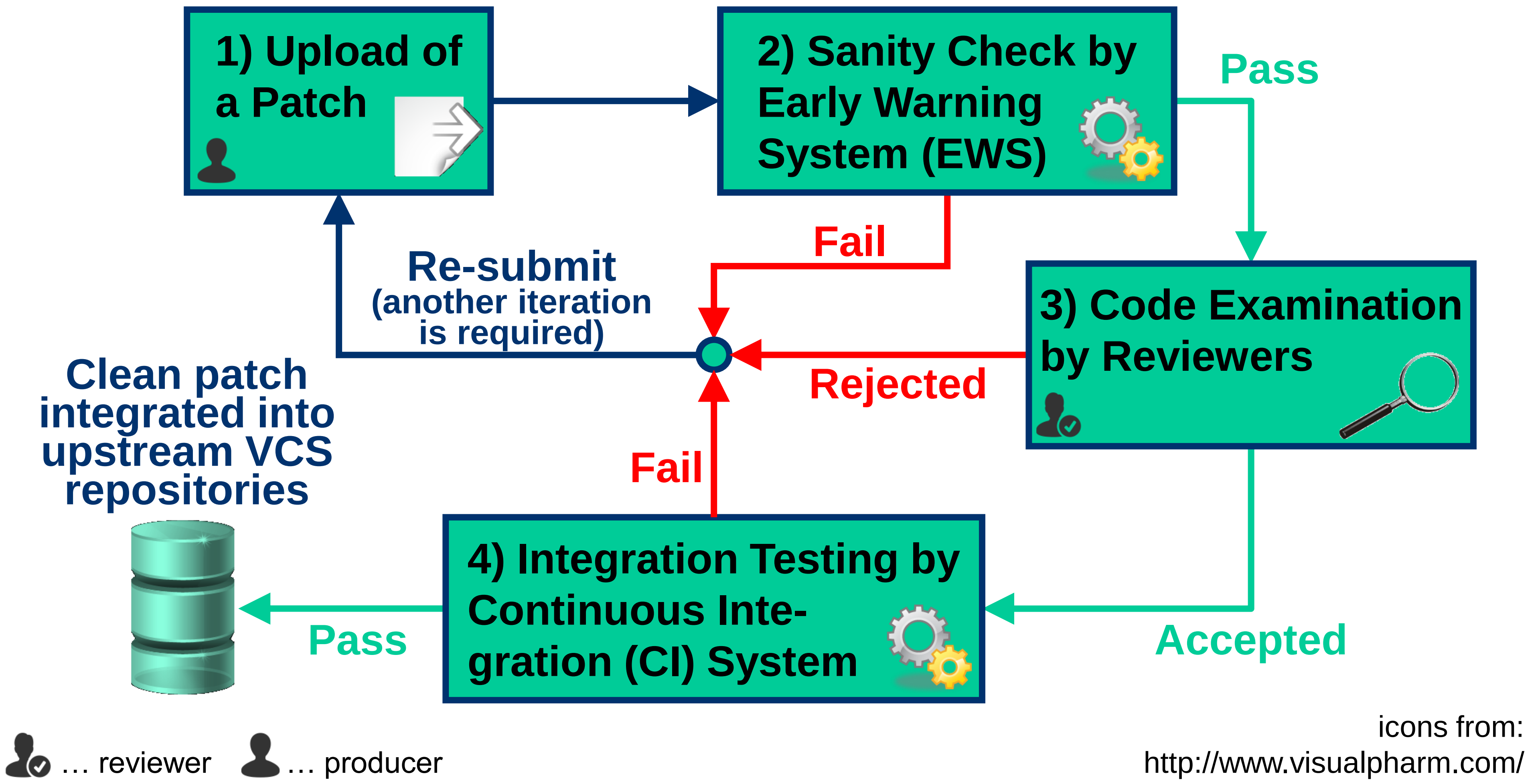
- **The Linux kernel and Apache:**
  - Low-tech reviews submitted to an email mailing list
  - Rely on the interest of the community to review interesting patches

- **Android, Eclipse, Firefox, Qt, Openstack:**
  - Use dedicated reviewing tools to perform reviews
  - The tools keep track of all review information
  - Automate the integration process

Modern Reviews: Roles

- **Author**:
  - Responsible for correcting problems that are identified during the review
- **Reviewer**:
  - Analyzes and detects problems in the artifacts
  - An engineer with expertise in the context that the artifact operates within
  - Often, at least two reviewers must agree with an artifact before it is deemed valid
- **Number of artifacts reviewed significantly higher than in traditional setting**

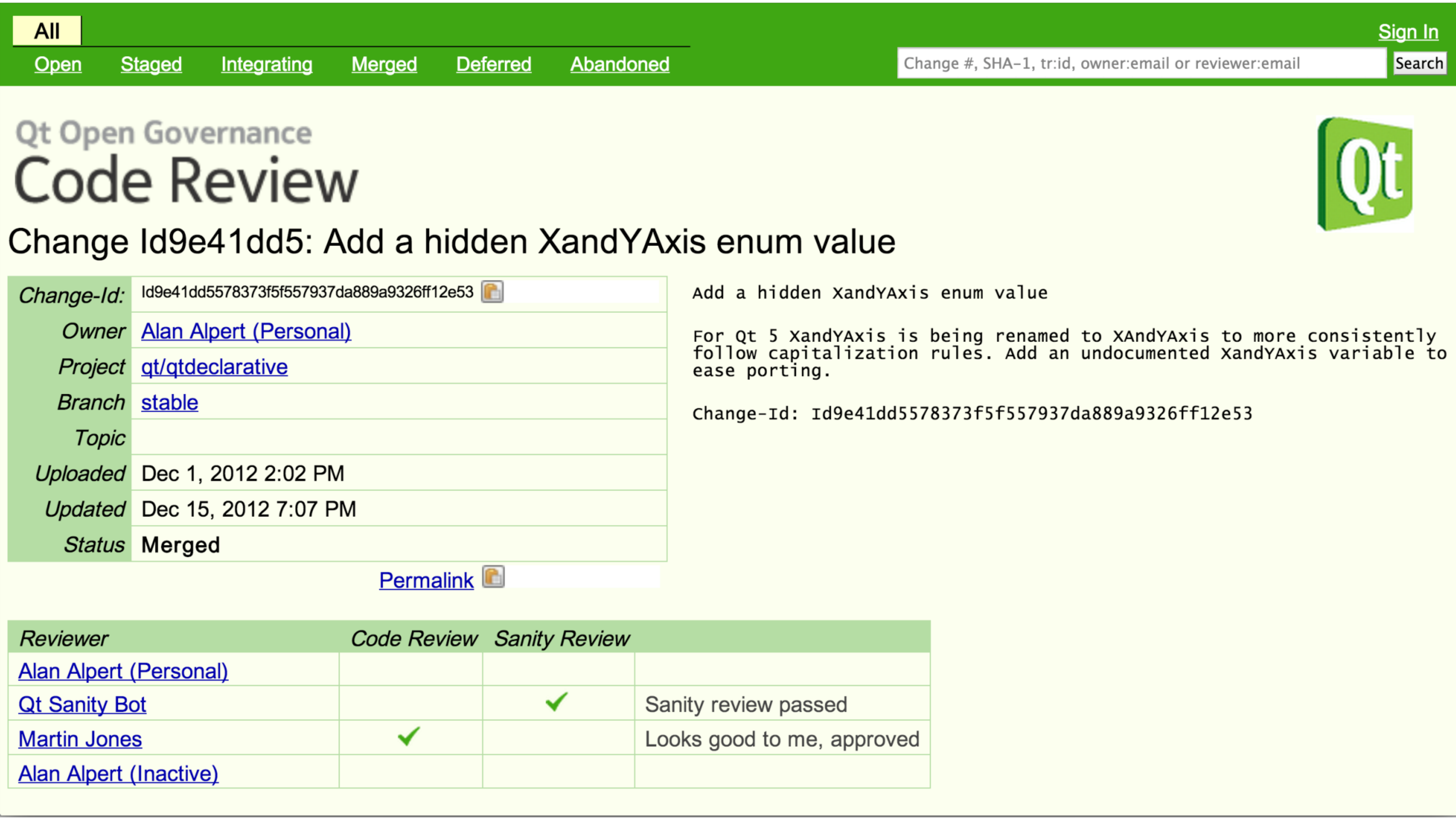
Example: Qt Code Review Process



1) Upload of a Patch
2) Sanity Check by Early Warning System (EWS)
Pass
Fail
3) Code Examination by Reviewers
Re-submit (another iteration is required)
Rejected
Fail
Accepted
4) Integration Testing by Continuous Integration (CI) System
Pass
Clean patch integrated into upstream VCS repositories
… reviewer
… producer

icons from: http://www.visualpharm.com/

Example Review: Qt Project



Change Id9e41dd5: Add a hidden XandYAxis enum value

Change-Id:	Id9e41dd5578373f5f557937da889a9326ff12e53
Owner	Alan Alpert (Personal)
Project	qt/qtdeclarative
Branch	stable
Topic	
Uploaded	Dec 1, 2012 2:02 PM
Updated	Dec 15, 2012 7:07 PM
Status	**Merged**

Permalink

```
Add a hidden XandYAxis enum value

For Qt 5 XandYAxis is being renamed to XAndYAxis to more consistently
follow capitalization rules. Add an undocumented XandYAxis variable to
ease porting.

Change-Id: Id9e41dd5578373f5f557937da889a9326ff12e53
```

Reviewer	*Code Review*	*Sanity Review*	
Alan Alpert (Personal)			
Qt Sanity Bot		✓	Sanity review passed
Martin Jones	✓		Looks good to me, approved
Alan Alpert (Inactive)			

Best Practices for Code Reviews

Guidelines by Palantir

Motivation

- Improve code quality and benefit from positive effects on team and company culture
- Keep commiters motivated
- Share knowledge
- Ensure consistency, legibility, compliance
- Avoid accidental errors

When to review?

- After automated checks completed successfully
- Before code merged to main branch
- Stacked CR model for complex changes
- Code reviews are „classless" (incl. even the most senior people)

https://medium.com/palantir/code-review-best-practices-19e02780015f

Preparing Code for Review

- **It is the author's responsibility to submit code that is easy to review (not to waste reviewers' time)**
  - Prefer small changes:
    - <5 files, <1-2 days to write, <20 mins to review
  - Only submit complete, self-reviewed, self-tested code
    - Code should pass (both locally and on CI server) before assigning reviewers
    - Refactoring changes should not alter behavior
  - Commit messages guidelines (next slide)
  - Finding reviewers:
    - Project lead or senior engineer

https://medium.com/palantir/code-review-best-practices-19e02780015f
https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html

Sample commit message

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

Performing Code Reviews

Purpose
- Does the code accomplish the author's purpose?
- Ask questions

Implementation
- How would you solve the problem?
- Potential useful abstractions?
- Use of existing libraries?
- New dependencies?

Legibility and Style
- Required reading effort?
- Adherence to coding style?
- Does this code have TODOs?

Maintainability
- Read the tests
- Backward compatibility?
- Need for integration tests?
- Feedback on documentation and comments

Security
- Verify API endpoints
- Authorization + Authentication
- Check for common weaknesses (e.g. Malicious user input)

- **Responsibilities of a reviewer:**
  - Review should be prompt (hours, not days)
  - Enforce coding standards
  - Keep quality high

How to Comment?

- **General guidelines: Concise, friendly, actionable**
- **Critique the code, not the author**
  - Avoid processive pronouns: “*your* method has a bug”
  - Avoid absolute judgements: “this can *never* work”
- **Differentiate between**
  - Suggestions: “Suggestion: extract method to improve legibility”
  - Required changes: “Add @Override”
  - Points that need clarification: “Is this really the correct behavior? If so, please add a comment explaining the logic.”
- **Once completed: Indicate**
  - Required response to comments
  - Need for re-review

See examples at: https://medium.com/palantir/code-review-best-practices-19e02780015f

Responding to Reviews

- **Respond to every comment**
  - Even if ACK or DONE
- **Explain why you made certain decisions**
- **If no agreement with reviewer**
  - Discuss in-person / real-time
  - Seek for outside opinion
- **Fixes should be pushed to the same branch but in separate commits**

10 Principles of a Good Code Review

1. If you commit to code review, review it thoroughly
2. Aim to understand every changed line
3. Don’t assume the code works – build and test it yourself (outside of unit tests)
4. Commenting matters
5. Review temporary code as strictly as production code
6. Consider how the code will work in production
7. Check documentation, tests and build files
8. Keep priorities straight when making suggestions (1. functional, 2. clean & maintainable 3. optimized)
9. Follow-up on reviews (i.e. review it again)
10. Reviewers are not perfect (e.g. ask for a 2nd reviewer)

https://dev.to/codemouse92/10-principles-of-a-good-code-review-2eg

Pressman's Inspection Guidelines

1) Review the product, not the producer
2) Set an agenda and maintain it
3) Limit debate and rebuttal (90-120min for meeting)
4) Identify problem areas, but problem solving should be postponed until after the review meeting
5) Take written notes (e.g., on a wall board)
6) Limit the number of participants and insist upon advance preparation
7) Develop a checklist for each product that is likely to be reviewed
8) Allocate resources and schedule time for reviews
9) Conduct meaningful training for all reviewers
10) Prepare report and establish follow-up procedure
11) Review the review process

Source: R.S. Pressman, 2000, Software Engineering – A Practitioner's Approach, 5th edition, McGraw Hill

Appendix A: Architecture Reviews

Architecture Reviews

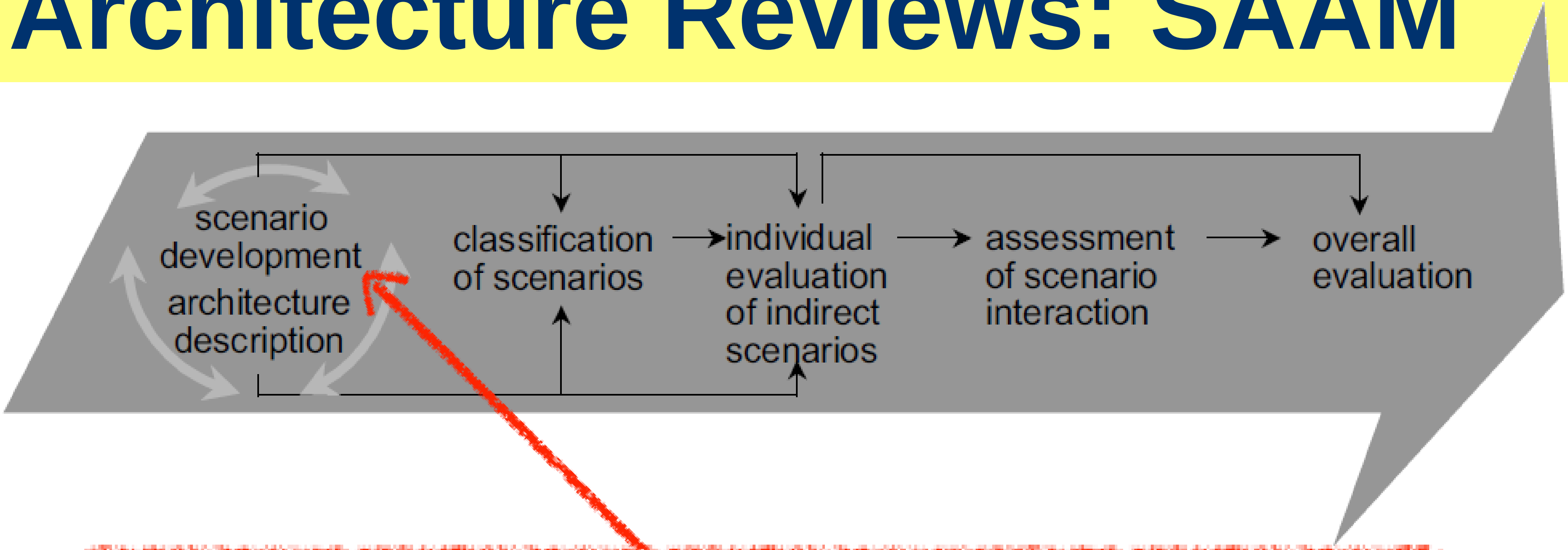
- **Goals:**
  - To evaluate an architecture regarding attributes such as maintainability and performance
  - To select among several possible architectural designs
- **Approaches:**
  - Using checklists (see Appendix B for more details)
  - Scenario-based techniques e.g.,
    - software architecture analysis method (SAAM)
    - architecture tradeoff analysis method (ATAM)
- **SAAM has seven steps**

Architecture Reviews: SAAM

1) Identify and assemble stakeholders

Stakeholder	Interest
Customer	Schedule and budget; usefulness of system; meeting customers' (or market's) expectations
End user	Functionality; usability
Developer	Clarity and completeness of architecture; high cohesion and limited coupling of parts; clear interaction mechanisms
Maintainer	Maintainability; ability to locate places of change
System administrator	Ease in finding sources of operational problems
Network administrator	Network performance; predictability
Integrator	Clarity and completeness of architecture; high cohesion and limited coupling of parts; clear interaction mechanisms
Tester	Integrated, consistent errorhandling; limited component coupling; high component cohesion; conceptual integrity
Application builder (if product line architecture)	Architectural clarity, completeness; interaction mechanisms; simple tailoring mechanisms
Domain representative	Interoperability

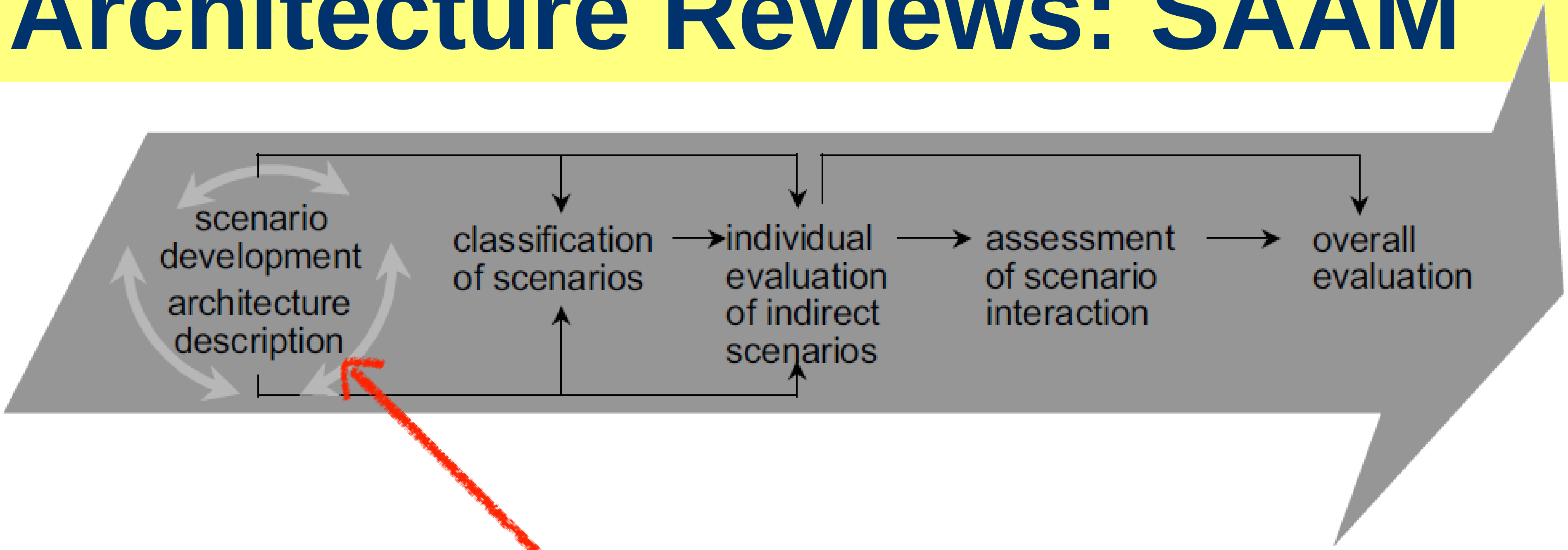
Architecture Reviews: SAAM


scenario development
architecture description
classification of scenarios
individual evaluation of indirect scenarios
assessment of scenario interaction
overall evaluation

2) **Develop and prioritize scenarios:**

- **Scenarios should be typical of the kinds of activities that the system must support:**
  - **Functionality**
  - **Development activities**
  - **Change activities**
- **Scenarios can also be chosen to give insight into the system structure**
- **Scenarios should represent tasks relevant to all stakeholders**

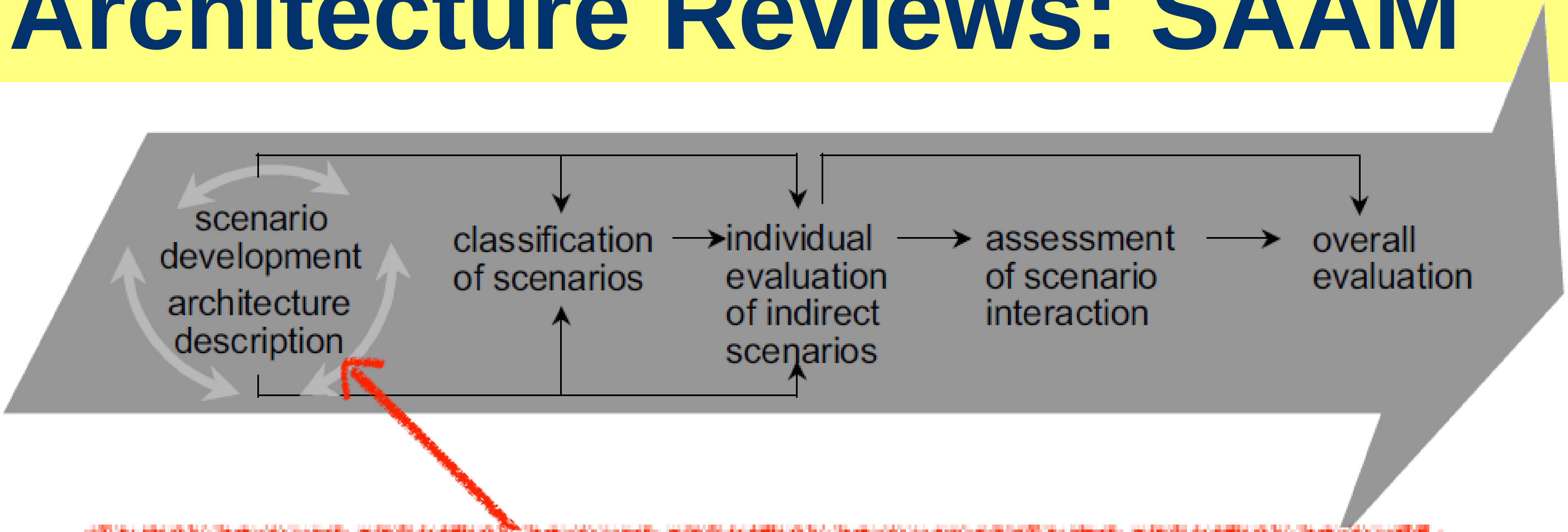
Architecture Reviews: SAAM


scenario development
architecture description
classification of scenarios
individual evaluation of indirect scenarios
assessment of scenario interaction
overall evaluation

3) Describe candidate architecture (if not done):

- It is frequently necessary to elicit appropriate architectural descriptions
- Structures chosen to describe the architecture will depend on the type of qualities to be evaluated:
  - Static structures will be used to evaluate modification scenarios
  - Dynamic structures will be used to evaluate run-time qualities

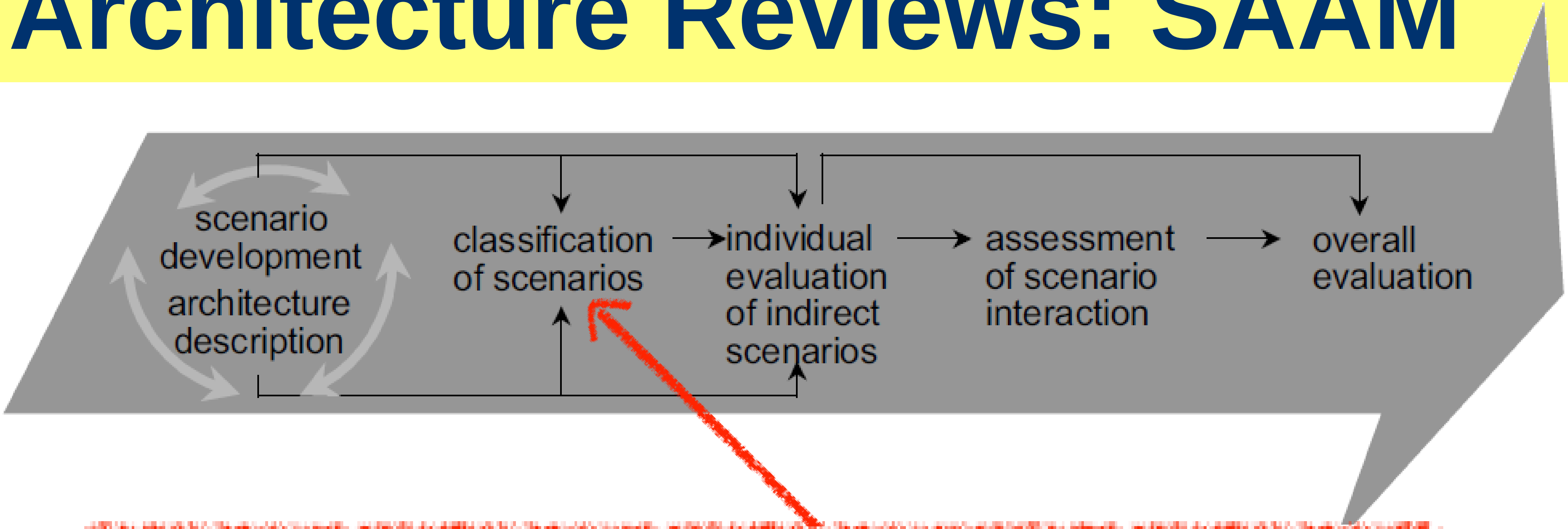
Architecture Reviews: SAAM


scenario development
architecture description
classification of scenarios
individual evaluation of indirect scenarios
assessment of scenario interaction
overall evaluation

3) Describe candidate architecture (if not done) (cont'd):

- Typically, the architect is asked to bring the following documentation to an evaluation, as a starting point
- A description of the logical or module structure, with a clear definition of the responsibilities of each component; a description of the process structure; a data flow diagram showing how the modules interact at run-time; a description of the interconnection mechanism(s) used to distribute data and control among architectural components at run-time; a description, if needed, of the "uses" structure, showing how potential subsets will be realized

Architecture Reviews: SAAM

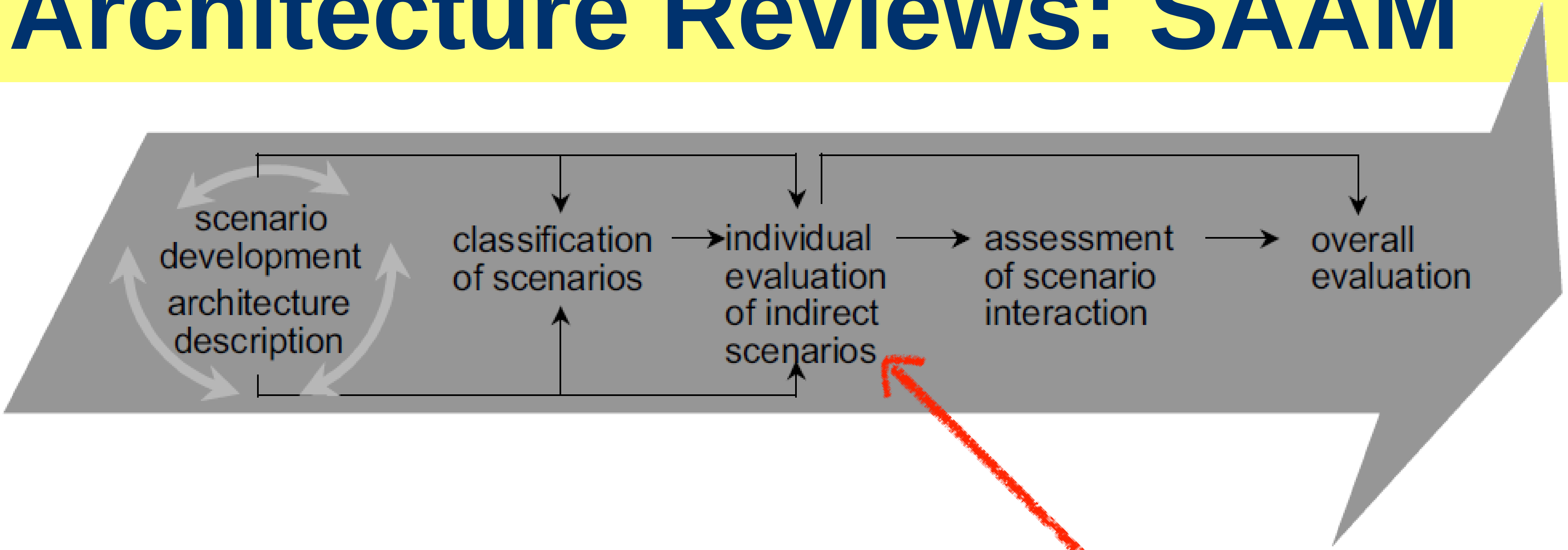


scenario development
architecture description
classification of scenarios
individual evaluation of indirect scenarios
assessment of scenario interaction
overall evaluation

4) Classify scenarios as direct or indirect:

- There are two classes of scenarios:
  - **Direct scenarios** are those that can be executed by the system without modification
  - **Indirect scenarios** are those that require modifications to the system
- The classification depends upon both the scenario and the architecture
- The classification should have **gradations**:
  - Indirect: how hard is the change?
  - Direct: how hard is it to execute?

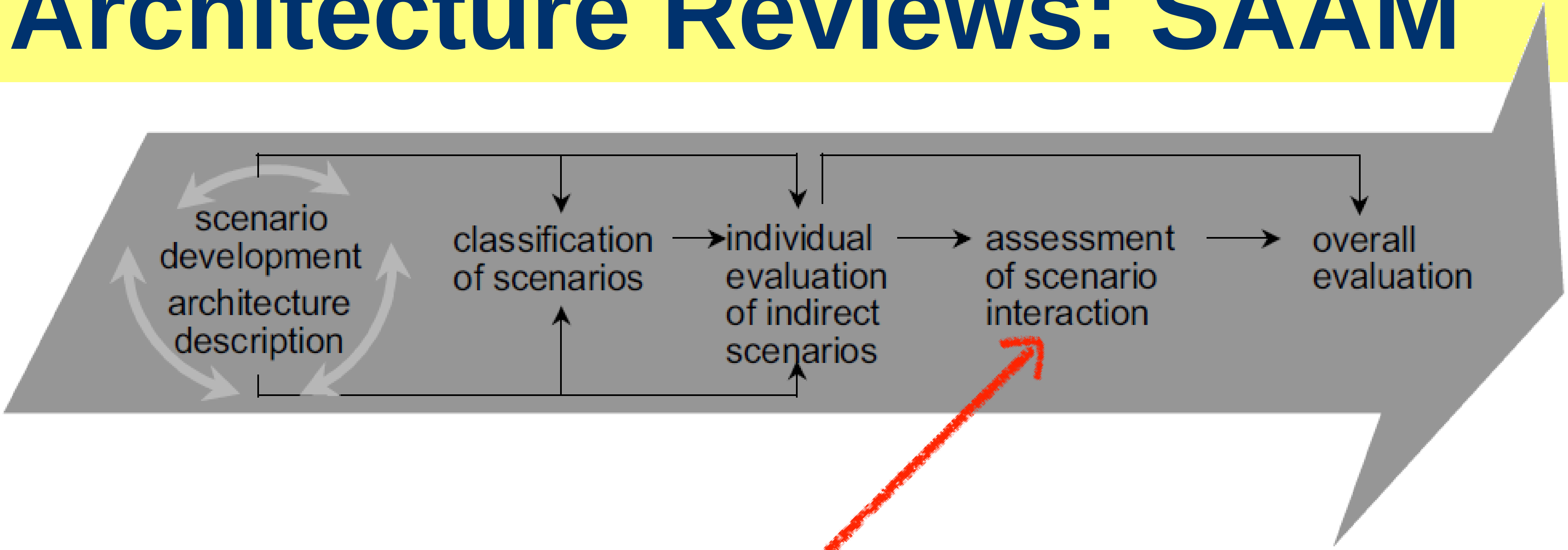
Architecture Reviews: SAAM


scenario development
architecture description
classification of scenarios
individual evaluation of indirect scenarios
assessment of scenario interaction
overall evaluation

5) **Perform scenario evaluation:**

- **For each indirect scenario:**
  - **Identify the components, data connections, control connections, and interfaces that must be added, deleted, or modified**
  - **Estimate the difficulty of modification**
- **Difficulty of modification is elicited from the architect and is based on the number of components to be modified and the effect of the modifications**
- **A monolithic system will score well on this step, but not on next step**

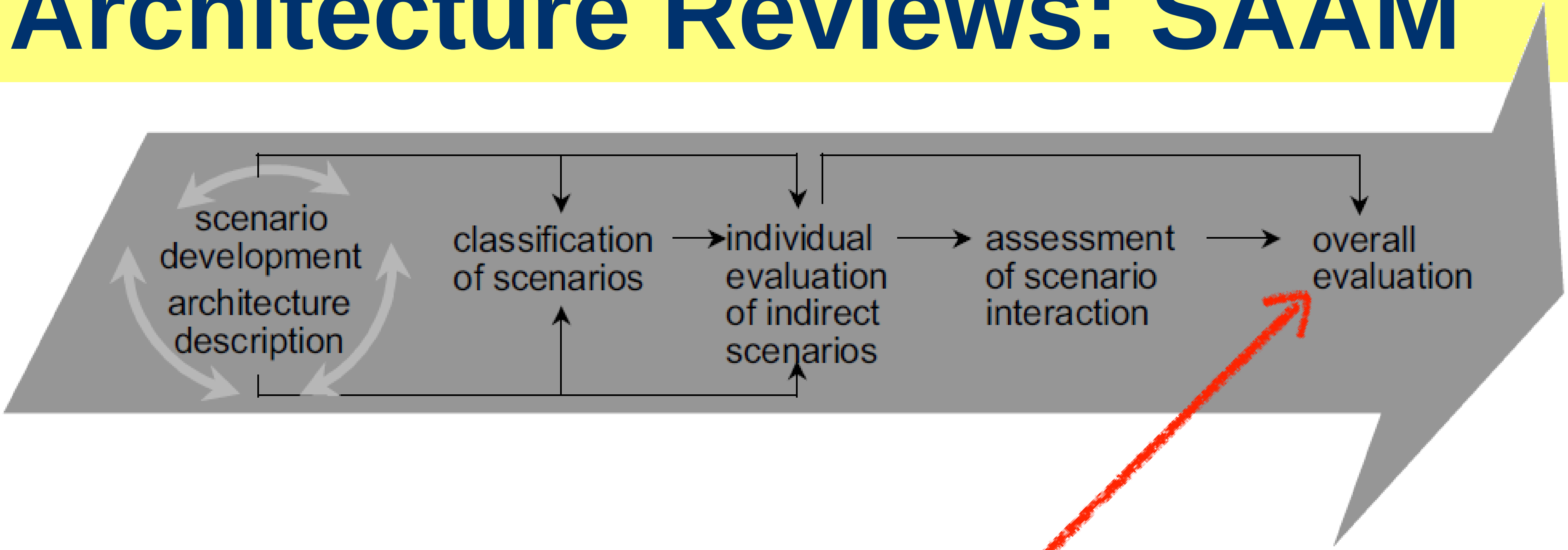
Architecture Reviews: SAAM


scenario development
architecture description
classification of scenarios
individual evaluation of indirect scenarios
assessment of scenario interaction
overall evaluation

6) Reveal scenario interactions:

- When **multiple indirect scenarios** affect the **same components**, this could indicate a problem:
  - Could be good, if scenarios are variants of each other (e.g., create a new account for customer, update account for customer)
  - Could be bad, indicating potentially poor separation of concerns (e.g., create a new account for customer, port system to a different platform)

Architecture Reviews: SAAM


scenario development
architecture description
classification of scenarios
individual evaluation of indirect scenarios
assessment of scenario interaction
overall evaluation

7) **Generate overall evaluation:**

- **Not all scenarios are equal**
- **The organization must determine which scenarios are most important**
- **Then, the organization must decide as to whether the design is acceptable "as is" or if it must be modified**

Architecture Reviews: SAAM

Exercise!

- **Potential review results of SAAM:**
  - Set of ranked issues
  - Enhanced system documentation
  - Set of scenarios for future use
  - Identification of potentially reusable components
  - Estimation of costs and benefits
  - Technical: provides insight into system capabilities
  - Social: forces some documentation of architecture; acts as communication vehicle among stakeholders
- **Most useful in evaluating non-run-time qualities (e.g., maintainability, flexibility…)**
- **Specifies context through scenarios (interaction of stakeholders with system – important to consider a comprehensive set of stakeholders)**

Appendix B: Checklists

Requirements Review

- **Making sure that users' needs are properly understood early**

- **Requirements can be:**
  - Formal or informal
  - Implicit or explicit

- **Look for:**
  - Definition of the required basic functionality
  - Definitions of important terms
  - Things that are vaguely defined

Requirements Review Checklist

- **Precision, unambiguity, clarity:**
  - Each item is exact and not vague
  - There is a single interpretation
  - The meaning of each item is understood
  - The specification is easy to read
- **Consistency**
  - No item conflicts
- **Relevancy**
  - Each item is pertinent to the problem and its eventual solution

Requirements Review Checklist

- **Testability**:
  - During program development and acceptance testing, it will be possible to determine whether the item has been satisfied
  - Generally suppose measurability

- **Traceability**:
  - During program development and testing, it will be possible to trace each item through the various stages of development

Functional Specification Review

- **Goals:**
  - To check how successfully the user requirements have been incorporated
  - Requirements document is used for checking the functional specification
  - There should be a justification when something in requirements documents cannot be traced to the functional specification
    - To check for incompleteness (what is missing?)
    - To check for unwanted additions

Functional Specification Review Checklist

- When a term is defined explicitly somewhere, try replacing occurrences of the term with its definition
- When a structure is described in words, try to sketch a picture of the structure being described
- When a calculation is specified, work at least two examples by hand and give them as examples in the specification
- Watch for vague words (e.g., some, sometimes, often, usually, ordinarily, customarily, most, mostly…)

Architecture Review Checklist

- **Review questions:**
  - Are software requirements reflected in the architecture?
  - Is effective modularity achieved?
  - Are modules functionally independent?
  - Are interfaces defined for modules and external system elements?
  - Is the data structure consistent with information domain?
  - Is data structure consistent with software requirements?
  - Has maintainability been considered?
  - Have quality factors been explicitly assessed?

Internal Design Review

- **Internal design includes:**
  - Detailed set of data structures, data flows, and algorithms
- **Review includes:**
  - Using checklists
  - Tracing back to architecture, functional specification (and requirements)
  - Seeing if there is an agreement with the algorithms
  - Looking for testing ideas

Internal Design Review Checklist

- Are the algorithms correct? Does the internal logic achieve the desired result?
- Is algorithm complexity low (e.g., McCabe's cyclomatic complexity should be less than 5)?
- Does each algorithm perform just a single task that could be coded in a few lines (roughly less than 30 lines)?
- Does the algorithm avoid redundant or duplicate code?
- Can a method be constructed from a sequence of other methods within the class?

Code Review

- Comparing code with internal design
- Examining code against language-specific checklist
- Using static analysis tool to check for compliance with the syntactic/content requirements
- Verifying correspondence of terms in code with data dictionary, internal design
- Searching for (may form basis for dynamic tests):
  - New boundary conditions
  - Possible performance bottlenecks
  - Other internal considerations
- Modern IDEs automate some of these checks
- Also read through the following blog post: http://swreflections.blogspot.ca/2014/08/dont-waste-time-on-code-reviews.html

Code Review Checklist

- **Data reference errors (using something that has not been properly declared or initialized):**
  - Is an uninitialized variable referenced?
  - Are array and string subscripts integer values?
  - Are array and string subscripts always within the bounds of the array's or string's dimension?
  - Are there any potential «off by one» errors in indexing operations or subscript references to arrays?
  - Is a variable used where a constant would actually work better (e.g., when checking the boundary of an array)?

Code Review Checklist

- **Data reference errors (cont'd):**
  - Is a variable ever assigned a value that's of a different type than the variable (e.g., assigning a float value to an integer variable)?
  - Is memory allocated for referenced pointers?
  - If a data structure is referenced in multiple functions or subroutines, is the structure defined identically in each one?

Code Review Checklist

- **Data declaration errors (improperly declaring or using something):**
  - Are all variables assigned the correct property, type, length (e.g., should variable be final; should variable be declared as string instead of array of characters)?
  - When a variable is initialized at its declaration, is it with a value consistent with its type?
  - Are there any variables with similar names (may be a sign that names have been confused)?
  - Are any variables declared that are never referenced or are referenced only once?
  - Are all variables explicitly declared within their specific module? If not, is it understood that the variable is shared?

Code Review Checklist

- **Computation errors (essentially math problems):**
  - Do any calculations that use variables have different data types, such as adding an integer to a float?
  - Do any calculations that use variables have the same data type but are different lengths (e.g., adding a byte to a word)?
  - Are the compiler's conversion rules for variables of inconsistent type or length understood and taken into account in any calculations?
  - Is the target variable of an assignment smaller than the right-hand expression?
  - Is overflow or underflow in the middle of a numeric calculation possible?
  - Is it ever possible for a divisor/modulus to be zero?

Code Review Checklist

- **Computation errors (cont'd):**
  - For cases of integer arithmetic, does the code handle that some calculations, particularly division, will result in loss of precision?
  - Can a variable's value go outside its meaningful range (e.g., could the result of a probability be less than 0% or greater than 100%)?
  - For expressions containing multiple operators, is there any confusion about order of evaluation and is operator precedence correct?
  - For expressions containing multiple operators, are parentheses needed for clarification?

Code Review Checklist

- **Comparison errors (susceptible to boundary condition problems):**
  - Are the comparison operators correct?
  - Are there comparisons between fractional or floating point values? If so will any precision problems affect the comparison?
  - Does each Boolean expression state what it should state? Does the Boolean calculation work as expected? Is there any doubt about the order of evaluation?
  - Are the operands of a Boolean operator Boolean (e.g., is an integer variable used in a Boolean calculation)?

Code Review Checklist

- **Control flow errors (usually caused by computation or comparison errors):**
  - Will the program, module, subroutine, or loop eventually terminate?
  - Is there a possibility of premature loop exit?
  - Is it possible that a loop never executes? Is it acceptable if so?
  - Can the index variable of a multiway branch statement (switch ... case) exceed the number of branch possibilities? If it does, is the case handled properly?
  - Are there any «off by one» errors that would cause unexpected flow or non-flow through the loop?

Code Review Checklist

- **Parameter errors (incorrect passing of data to and from functions, methods…):**
  - Do the types and sizes of parameters received match those sent? Is the order correct?
  - If constants are ever passed as argument, are they accidentally changed in called function/method…?
  - Does a function/method… alter a parameter that is intended only as an input value?
  - Do the units of each parameter match the units of each corresponding argument (e.g., English vs metric system)?
  - If global variables are present, do they have similar definitions and attributes in all referencing functions/methods…?

Code Review Checklist

- **Input/output errors**:
  - Does the software strictly adhere to the specified format of the data being read or written by the external device?
  - If a file or peripheral is not present or ready, is that error condition handled?
  - Does the software handle the situation of the external device being disconnected, not available, empty or full during read or write?
  - Are all conceivable errors handled by the software in an expected way?
  - Have all error messages been checked for correctness, appropriateness, grammar, and spelling?

Code Review Checklist

- **Other checks:**
  - Will the software work with languages other than English? Does it handle extended ASCII characters? Does it need to use Unicode instead of ASCII?
  - If the software is intended to be portable to other compilers and CPUs, have allowances been made for this?
  - Has compatibility been considered so that the software will operate with different amounts of available memory, different internal hardware such as graphics and sound cards, and different peripherals such as printers and modems?
  - Does compilation of the program produce any «warning» or «informational» messages?

Unit Testing Checklist

- For more details, see Unit Testing Checklist from DZone in the first JUnit tutorial