The software development life-cycle (SDLC) & security touch points

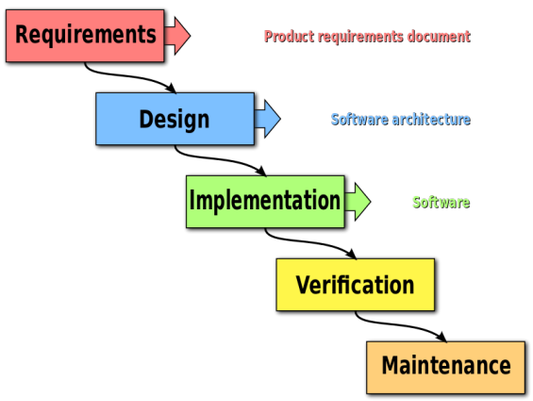
Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt

The SDLC

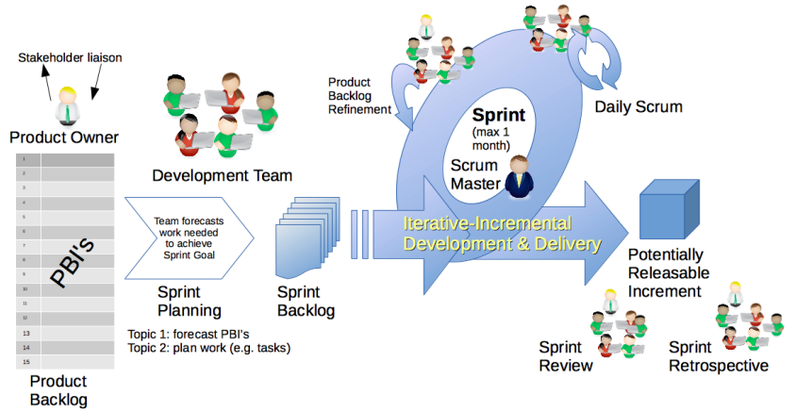
Images from Wikipedia

Waterfall



XP

SCRUM



- You probably heard about different types of software development methodologies.
- They are not just about *writing code*. Instead, they account for the **entire software development life-cycle (SDLC)** — that defines how to *engineer software*.
- Are they really needed ? Does the SDLC need to be disciplined/systematic?
  - In a word: yes.
  - Software plays a crucial role in everyday life. It needs to be *engineered* properly. Quite frequently, that is not the case.

SDLC stages

- Whatever the software system / development at stake, a SDLC typically considers:
  - **Requirements** — What should the software do ?
  - **Design** — How should it be structured ?
  - **Implementation** — Implementing the actual system.
  - **Verification —** Ensuring (with some degree of confidence) that the implemented system meets the requirements. This is usually done through *testing*.
  - **Deployment/maintenance** — Setting up the system for use and maintaining it.
    - During the semester we will mostly focus on aspects of implementation and verification from a security perspective.
- Some old methodologies like the “waterfall model” view these stages in sequence. In modern methodologies, the SLDC works more like a feedback loop.
- **In any case, where does security fit in ?**

Security touch-points

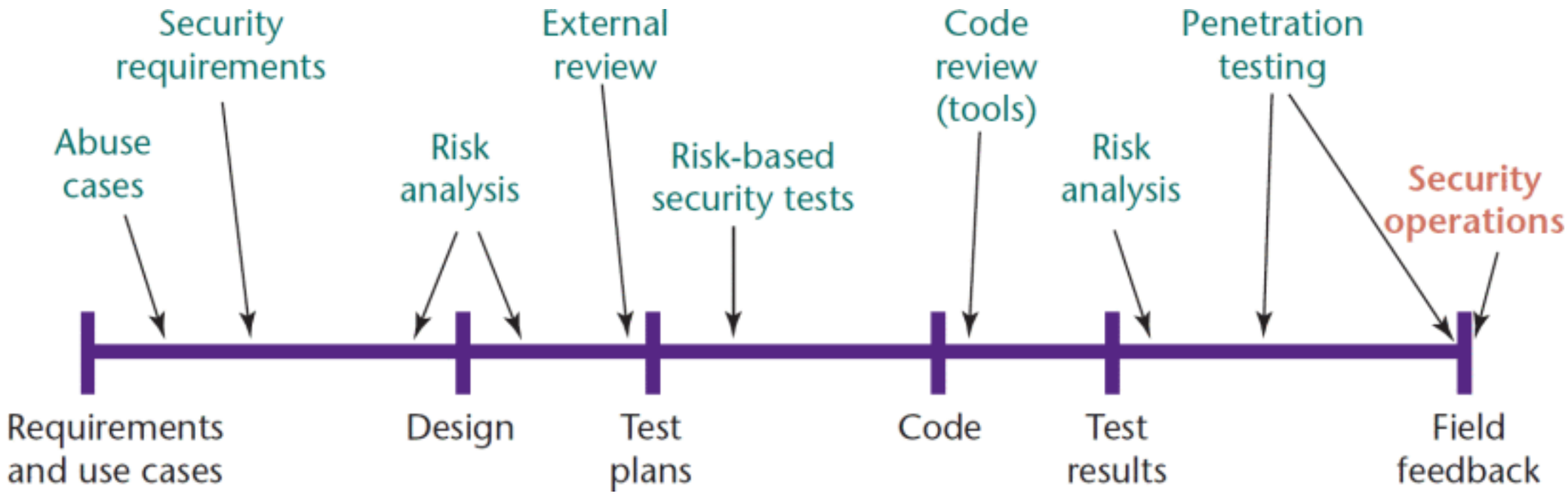


Image source: “Software security”, G. McGraw, IEEE SECURITY & PRIVACY

- **Security is a *cross-cutting* concern and an *emergent* property. It must be accounted for during the entire SDLC.**
- With that in mind, Gary McGraw proposed the influential touch-point model. The idea is that touch-points define security-oriented tasks for different stages of the SDLC.

Software requirements

Requirements

- **Software requirement**
  - Description of an expected feature of the software.
  - It must be: *"verifiable as an individual feature as a functional requirement or at the system level as a nonfunctional requirement*" - IEEE SWEBOK guide
- **Functional requirements**:
  - ***Feature = function of the system***
  - typically some kind of abstract input-output relation: *given X, compute f(X)*
  - Silly example: *given X calculate X + 1*"
- **Non-functional requirements**
  - ***Feature = constraint on the system***
  - Example constraints: resource (memory, bandwidth, …) quality-of-service guarantees (latency, throughput), environment setting
  - Silly examples: "the calculation of X+1 MUST be done in less than 1 second" (constraint on latency) or "without spending more than 4 GB of memory" (constraint on memory usage)

Requirements engineering

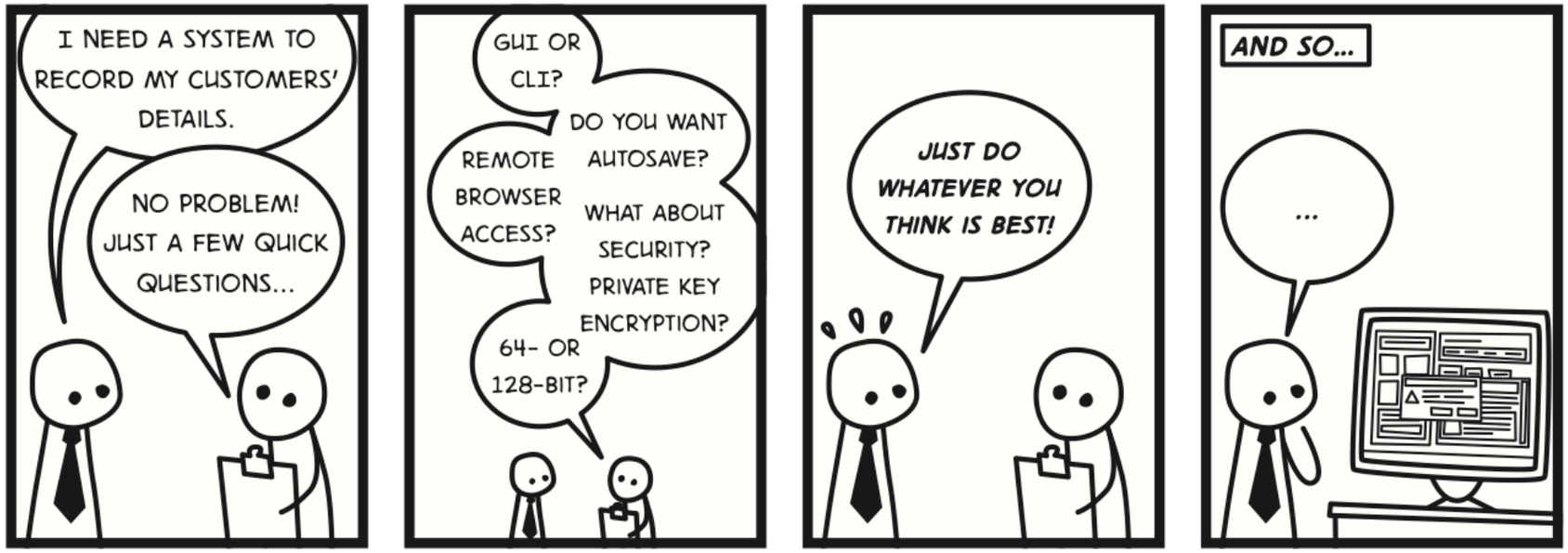


Image source:
Computer Science Field Guide

- Deriving requirements can be a process in itself, called ***requirements engineering***.
- For instance, the (dense) IEEE SWEBOK guide describes the following activities for requirements engineering:
  - **1. Elicitation**: Feedback is first collected from all stakeholders (people with an interest in the software), clients, users, consortium partners, the technical development team
  - **2. Analysis**: Requirements are then identified, possibly employing a number of modelling methodologies.
  - **3. Specification**: requirements are formalised in documents.
  - **4. Validation**: the specification is validated and subject to approval.

Example requirements — RFC style

5.1.1 Method

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

```
    Method         = "OPTIONS"                ; Section 9.2
                   | "GET"                    ; Section 9.3
                   | "HEAD"                   ; Section 9.4
                   | "POST"                   ; Section 9.5
                   | "PUT"                    ; Section 9.6
                   | "DELETE"                 ; Section 9.7
                   | "TRACE"                  ; Section 9.8
                   | "CONNECT"                ; Section 9.9
                   | extension-method
    extension-method = token
```

The list of methods allowed by a resource can be specified in an Allow header field (section 14.7). The return code of the response always notifies the client whether a method is currently allowed on a resource, since the set of allowed methods can change dynamically. An origin server SHOULD return the status code 405 (Method Not Allowed) if the method is known by the origin server but not allowed for the requested resource, and 501 (Not Implemented) if the method is unrecognized or not implemented by the origin server. The methods GET and HEAD MUST be supported by all general-purpose servers. All other methods are OPTIONAL; however, if the above methods are implemented, they MUST be implemented with the same semantics as those specified in section 9.

- From: RFC 2616 - HTTP/1.1
- Note the capital “MUST”, “SHOULD”, “OPTIONAL” …

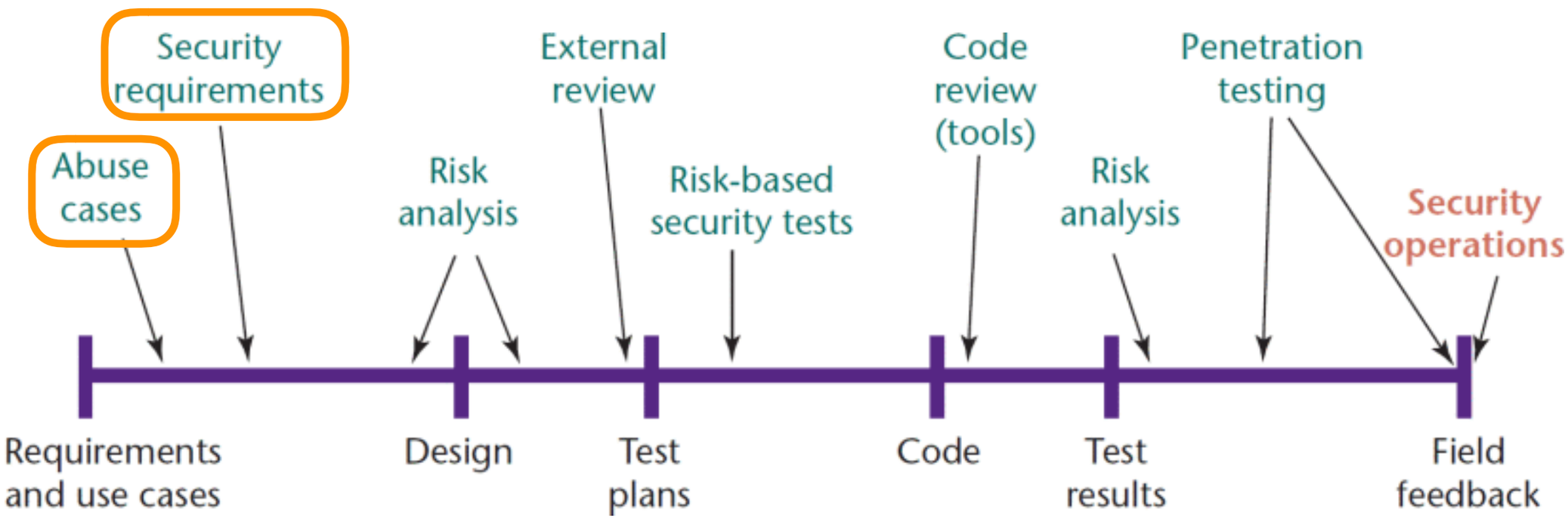
- RFC documents typically include this section to explain how to interpret requirements and compliance with requirements.

> **1.2 Requirements**
>
> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [34].
>
> An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be "conditionally compliant."

- RFC 2119 in turn specifies (fragment):

> 1. **MUST** This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.
>
> 2. **MUST NOT** This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.
>
> 3. **SHOULD** This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
>
> 4. **SHOULD NOT** This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

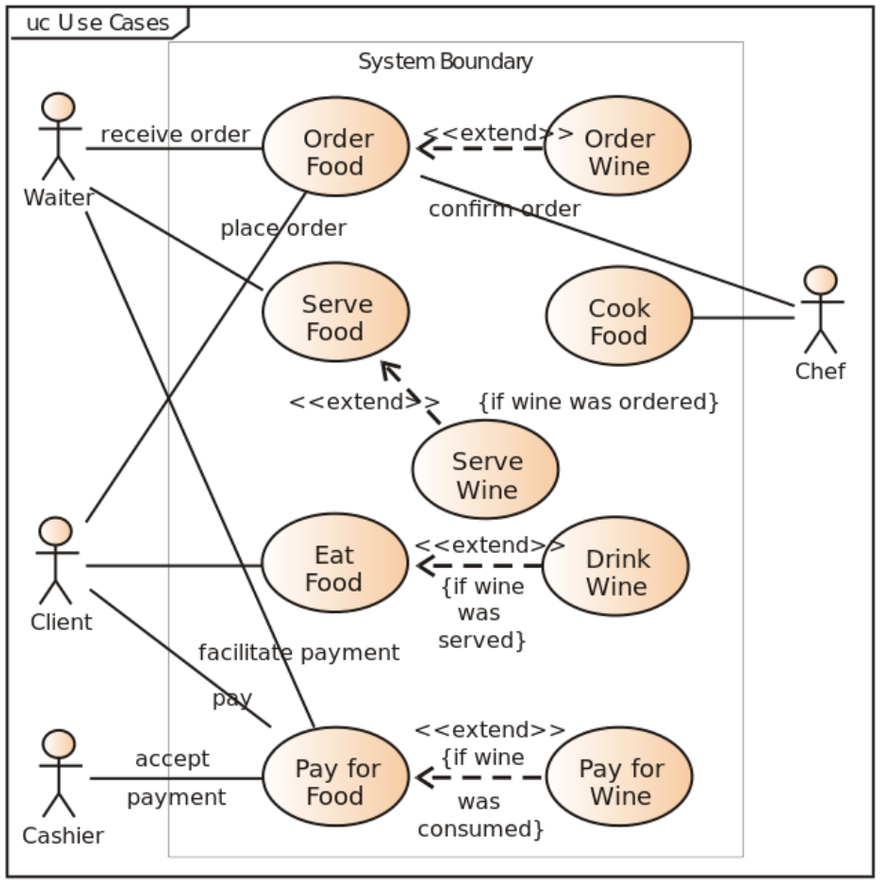
Security requirements

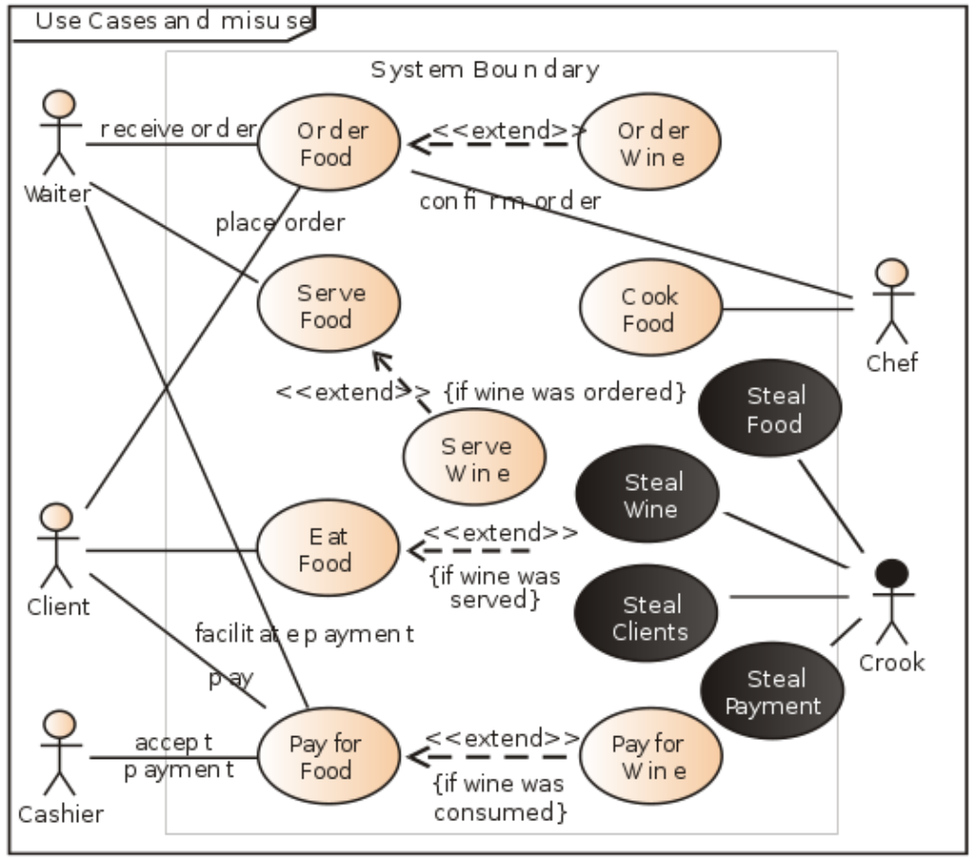


- **Security requirements** are driven by goals in the security policy:
  - Recall from last class: they may relate to confidentiality, integrity, availability, privacy, non-repudiation, …
- Identifying requirements in general may be driven by **use cases**, that model how a system may be *used*.
- Identifying security requirements may also be driven by **abuse cases,** that model how a system may be *abused.*

Use cases / Abuse cases

Images from Wikipedia





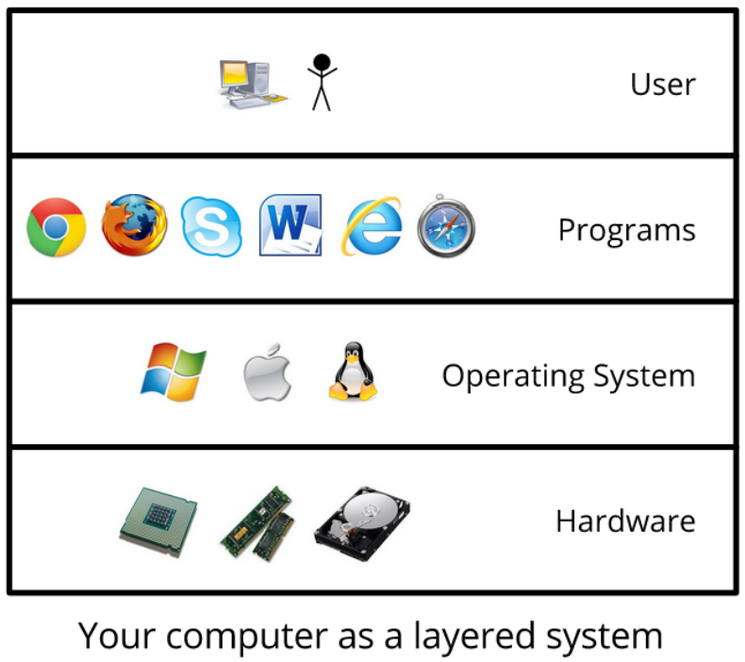
- **Use case:**
  - High-level description of interactions among actors in a system, modelling normal use.
- **Abuse case** (also called misuse case)
  - Like an use case, but focused on modelling adversarial actions as well.
- **Aids**:
  - Modelling languages with a visual syntax like UML or LML may help (a bit; see discussion next).

Deriving abuse cases

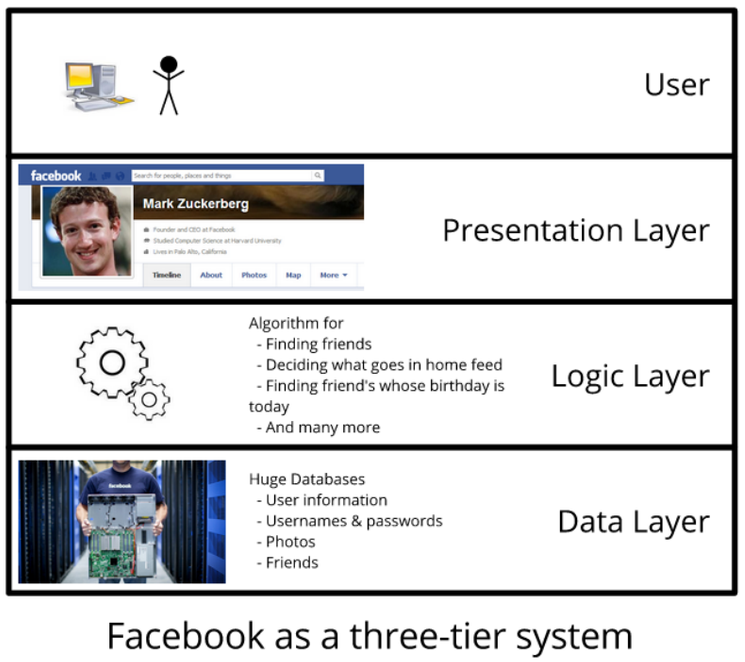
- Some general advice (from "Misuse and Abuse Cases: Getting Past the Positive" by Hope, McGraw, and Antón)
  - "**Informed brainstorming**" rather than the use *of "rigorous formal models and logics [that] are extremely time and resource intensive."*
  - **Threat modelling**: *"security experts ask many questions of a system's designers to help identify the places where the system is likely to have weaknesses. This activity mirrors the way attackers think."*
    - Taxonomies like STRIDE help in this regard .
  - Consider **Attack patterns ("***blueprint[s] for creating an attack.")*
    - CAPEC (Common Attack Patterns Enumeration and Classification)
    - The list of attack patterns *from* "Exploiting Software: How to break code" by G. McGraw
    - Attack pattern modelling definitions *in US-CERT website.*

Software design

Software design



Your computer as a layered system

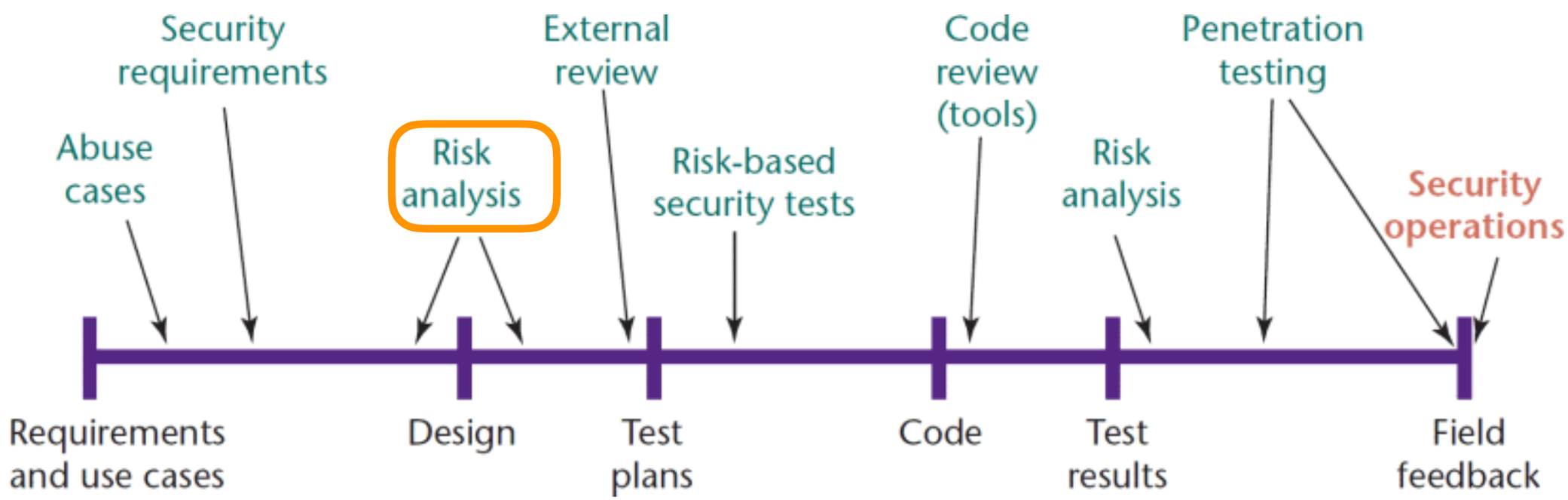


Facebook as a three-tier system

Image source:
[Computer Science Field Guide]

- **Software design** seeks to define a model for a software system.
- Some basic aspects of design:
  - **Decomposition**: identifying necessary components and their modular role.
  - **Architecture**: define how components assemble together and interact, often organised in layers.
  - **Abstraction & refinement:** to be able to model behavior as abstract at high level, and then refine it / instantiate it in more detail at a lower level of design.
  - **Data types and functions:** model types of data and associated functionality.
- ***All these important for code too … design and implementation are often blurred.***

Design — architectural risk analysis



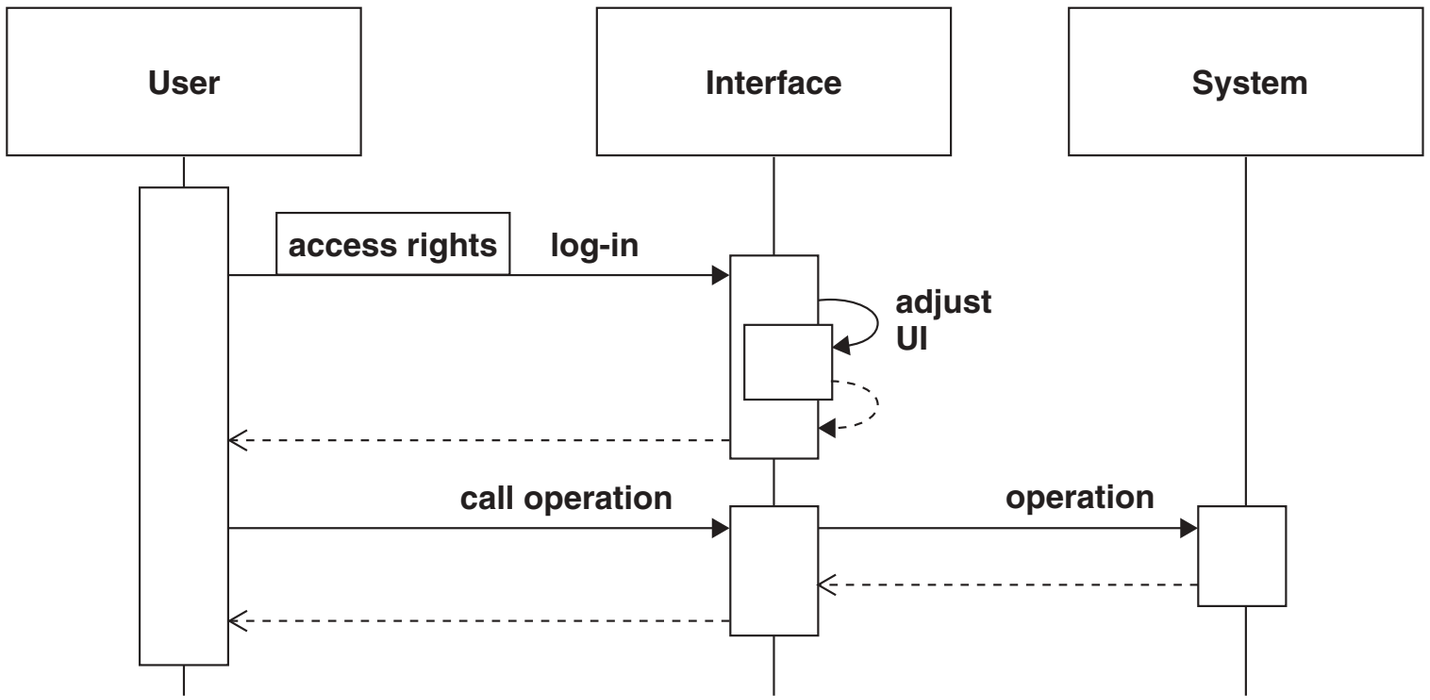
- **Architectural Risk analysis (ARA)**
  - Assess a architecture with respect to security risks.
- **Aids**:
  - Model threats and their risk — recall the STRIDE and DREAD models from last class — accounting for the system architecture, dependencies, roles, …
  - Design principles
  - Design patterns for security

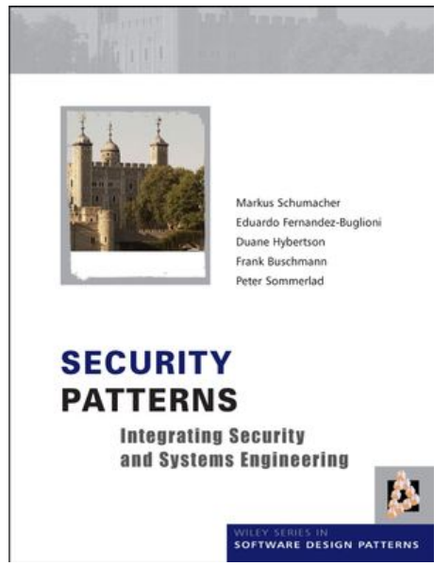
Design principles

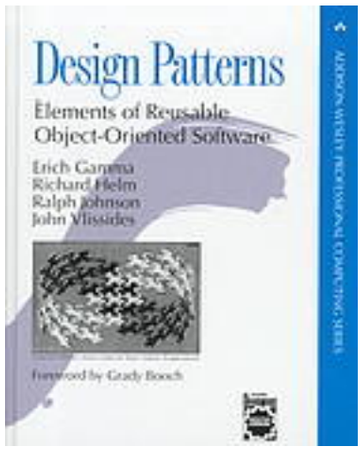
- **1. Secure the weakest link**
  - Security defense should be seen as a chain, attackers will look for the weakest link in that chain.
- **2. Defense in depth**
  - Manage risk by defense at all layers / components, such that if one of them fails, the other has a fair chance.
- **3. Fail securely**
  - Handle failures correctly & securely; failure handling is often overlooked in reliability and security terms.
- **4. Least privilege**
  - Execute software with the minimum required privileges to mitigate possible impact of an attack (e.g. do not run servers with super-user privileges)
- **5. Compartmentalize**
  - Try to limit the damage by compartmentalizing (e.g. using VPNs, containers, firewalls …)
- **6. Keep It Simple (KISS) !!!**
- **…**

[Viega and McGraw, Building Secure Software, chapter 5]

Design patterns



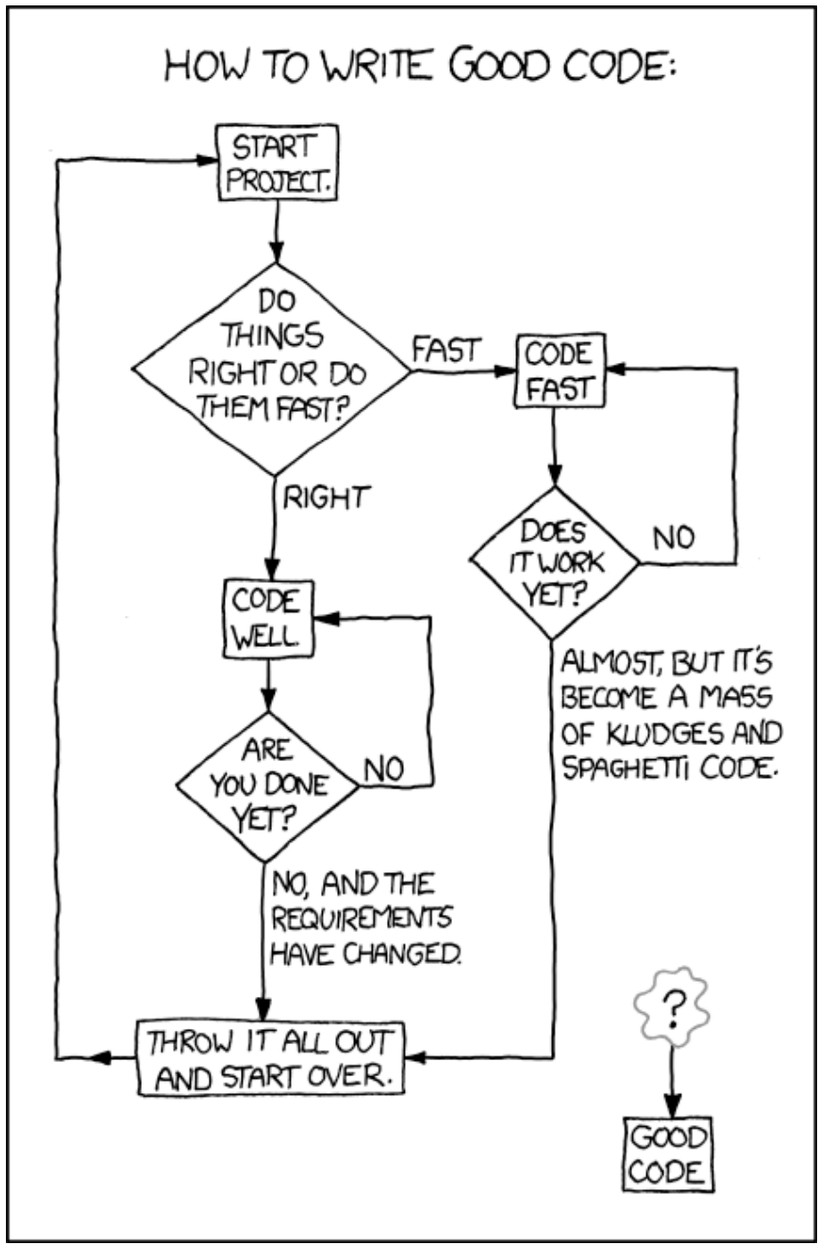




- **Design patterns**
  - description of a general solution to a recurrent problem
  - unlike design principles: concrete formulation, specific problem
  - the "Gang of Four" book has been quite influential
- **Security patterns** have been be defined in the same spirit
  - Check “Security Patterns: Integrating Security and Systems Engineering”, by Schumacher et al.

Software implementation

Discussion



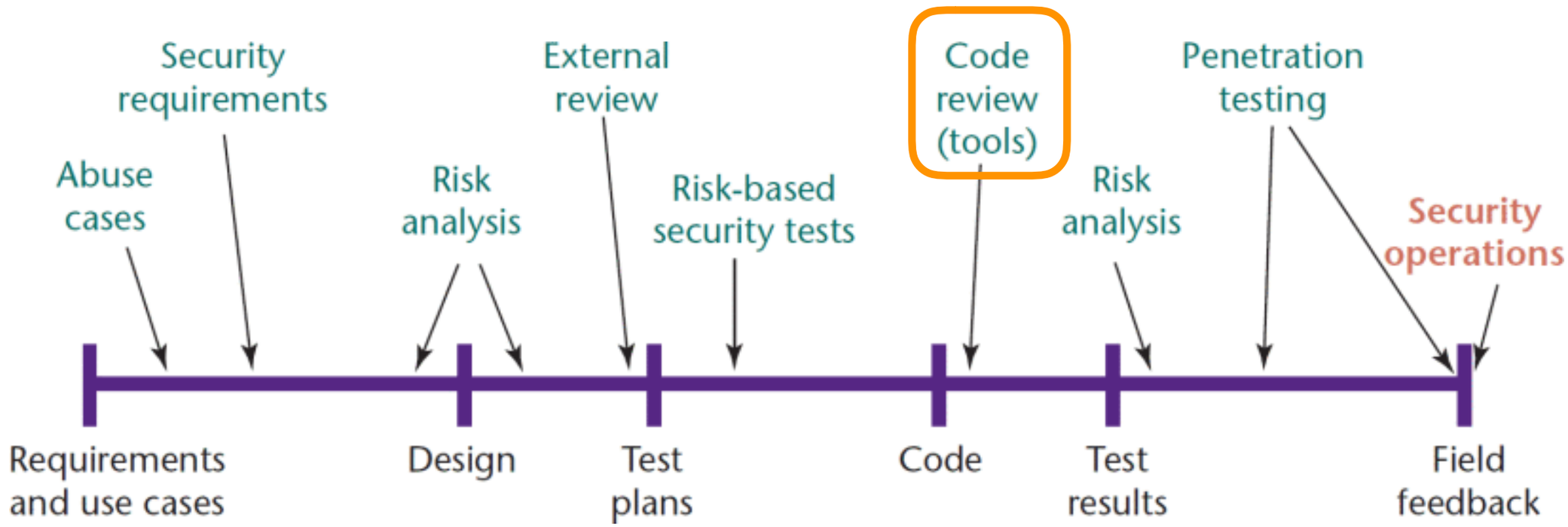
xkcd: [Good Code]

- What do you think …
  - … is good code?
  - … can someone be confident about it? How do you trust your own code or someone else's?

Good code

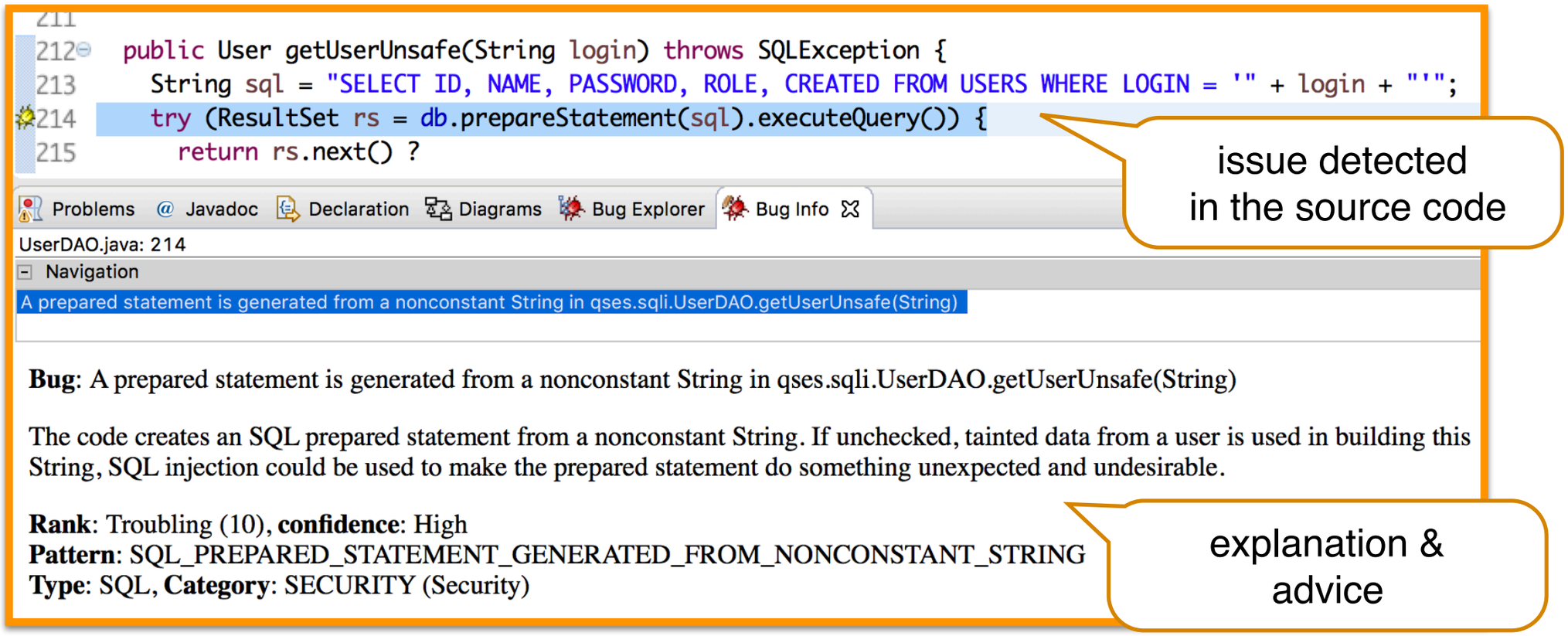
- Some key desirable properties for code … (what have we been talking about after all?):
  - *Reliable*: does what it supposed to.
  - *Secure*: … and nothing else.
  - *Flexible*: composes well in interaction with other components, may be used in different environments.
  - *Verifiable*: can be tested properly or verified / analysed by other means
  - *Maintainable*: is adaptable to change over time and easy to understand & maintain by others.
- How to get some level of confidence in the quality (and in particular the security) of my code ?
  - Code reviewing (next)
  - Verification (the most common approach is testing, covered also in these slides)

Code review



- Code reviewing
  - The activity of auditing code.
  - Strictly involves the (static) code only! No execution!
- General aim — identify problems with early in the software development:
  - enforce coding standards
  - finding software defects (code location of standard bugs) ***and security vulnerabilities***
  - to identify problems in code during the implementation stage.

Static analysis tools & code reviewing



Screenshot: SpotBugs + FindSecBugs in action (Eclipse IDE)

- Operation:
  - **1)** Static analysis is configured by a set of rules & associated parameters.
  - **2)** The code is analysed automatically.
  - **3)** A human-readable report is produced for all the problems found in the code.
- Good things
  - Exact locations of potential problems are spotted.
  - Tools also almost always provide explanation & advice on code changes.

Static analysis tools & code reviewing

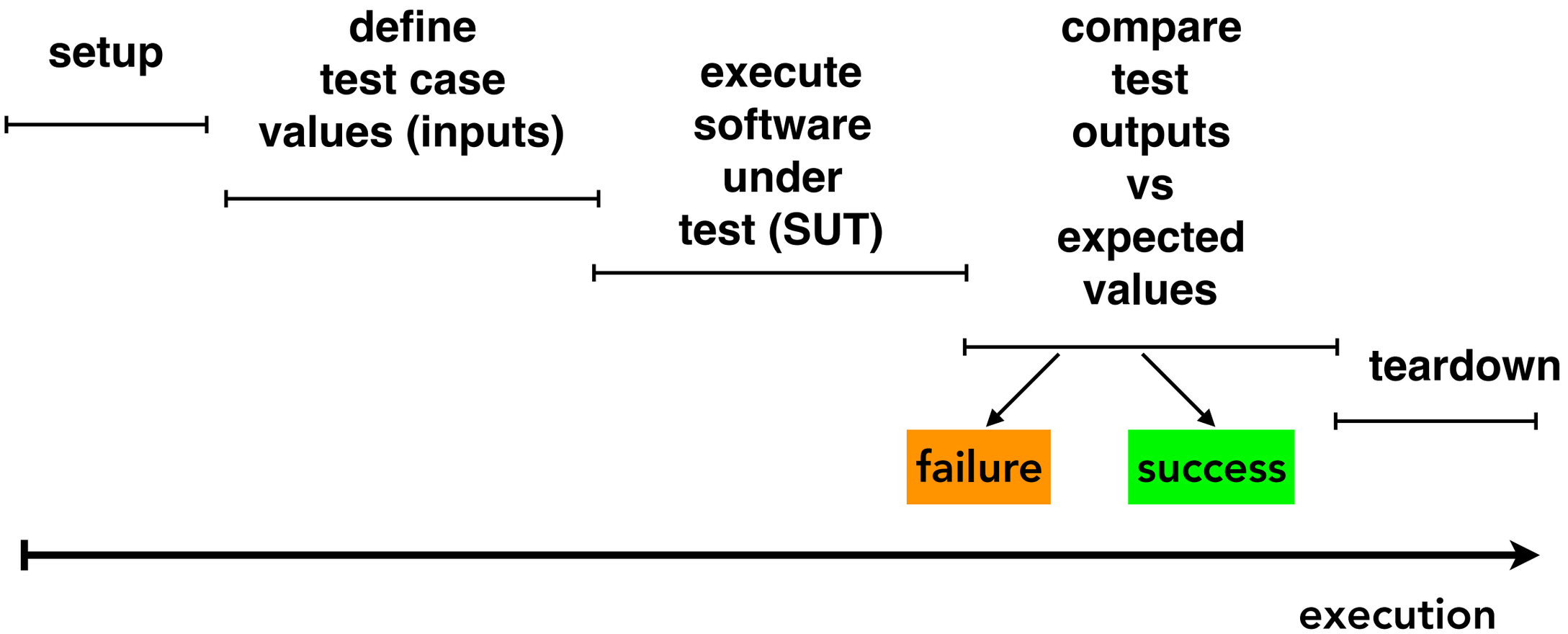
- Keep in mind — lack of soundness & completion:
  - *The analysis will not always be sound*. “False positives” arise.
  - *The analysis will not be complete*. It will not detect all problems.
- Developer must then perform the actual code review (the tools just help!):
  - verify if detected issues are real problems to fix in the code or instead false positives …
  - … and examine issues not covered by static analysis.
- In particular, code review meetings may be held during development.

Software Testing

Software testing

- **Testing**
  - Observe if software meets the expected behavior when **executed**.
  - Does not guarantee absence of bugs, in fact it seeks to *expose them.*
- Testing vs static analysis
  - When a test fails, it does not identify the defect(s) in source code, only their manifestation.
  - No false positives.

Test case



- A **test case** has the following ingredients:
  - The **subject-under-test** (SUT), e.g. function, class, component.
  - The **inputs for the test** and the **expected outputs**.
  - Also, optional setup/teardown actions (we won't go into detail for now).

Test case programming — xUnit style

```
@Test
public void testNumZeroArrayWithNoZeros() {
   int[] x = { 1,2, 3 };
   int n = numZero(x);
   assertEquals(0, n);
}
   ...
```

1) setup test case values

2) execute SUT (numZero)

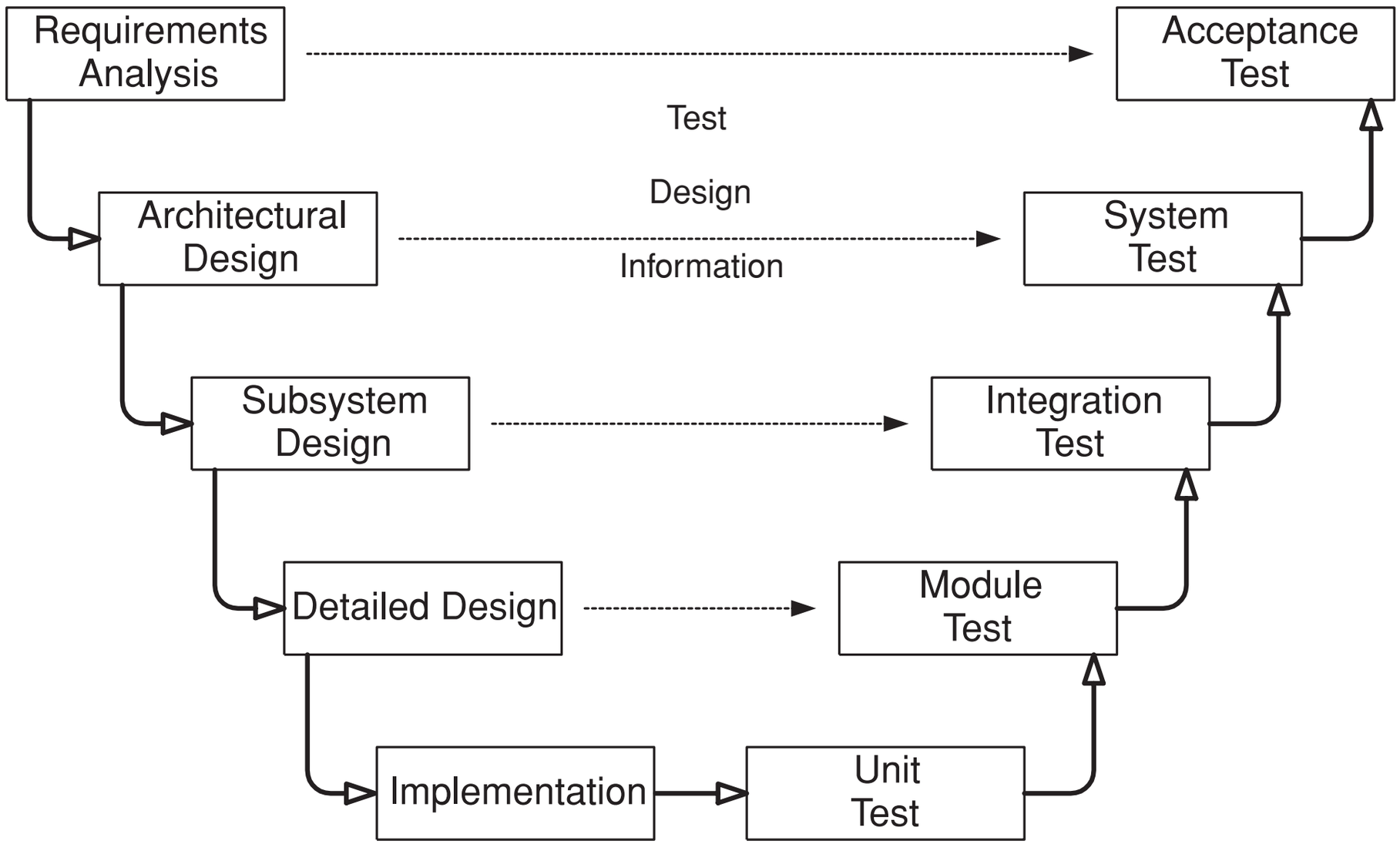
3) assert expected vs. test outputs

- A simple example in Java using JUnit
- So tests take form as code too …
  - that need to be (well) coded, maintained, etc

Remarks on testing

- Black-box testing vs white-box testing
  - Black-box: tests are based on the abstract requirements, not the structure of the code
  - White-box: tests are guided by the structure of the code
- How to assess the quality of a test suite?
  - Some criteria is required, that should translate to a precise metric. For instance:
    - **Line coverage** = % of lines of code that are exercised by tests (a “white-box” metric)
    - **Requirements coverage** = % of requirements tested (a “black-box” metric)
- Tests should be reproducible
  - Non-determinism may arise in a fragile test environment or inherent to the software itself
    - “Did you just delete that database entry I inserted yesterday for testing purposes?”
    - Concurrent code (e.g. multi-threaded code) is prone to non-determinism. Infamous “heisen-bugs” arise: they happen in particular circumstances that are hard to replicate.

Testing levels & the SDLC



From: “Introduction to Software Testing”, Amman & Offutt

How important is testing?

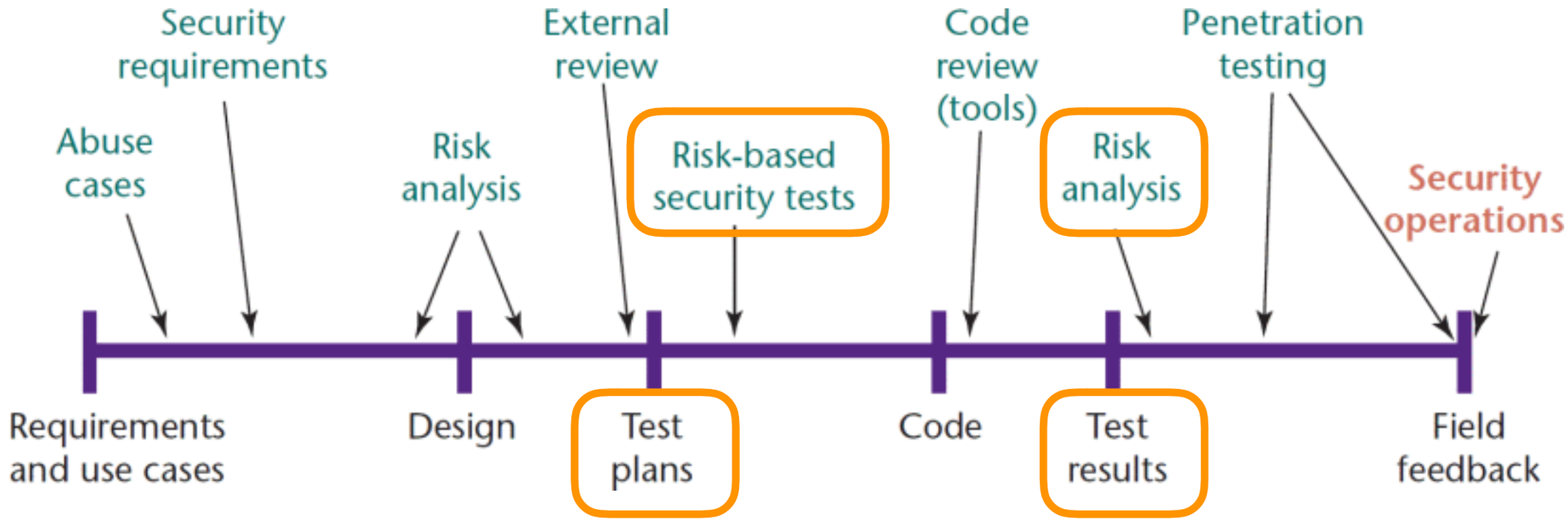
Image source: [History of software testing](), blog article, Ashish Singh, 2012

- Testing is the standard approach for ensuring that software has a high level of reliability.
- No (serious) SDLC process goes without testing.

How “mature” should testing be?

- **Beizer’s scale for test process maturity**
  - **Level 0:** “There’s no difference between testing and debugging.”
    - **Question: What is debugging?**
  - **Level 1:** “The purpose of testing is to show that the software works.”
  - **Level 2:** “The purpose of testing is to show that the software doesn’t work.”
  - **Level 3:** “The purpose of testing is not to prove anything specific, but to reduce the risk of using the software.”
  - **Level 4:** “Testing is a mental discipline that helps all IT professionals develop higher quality software.”

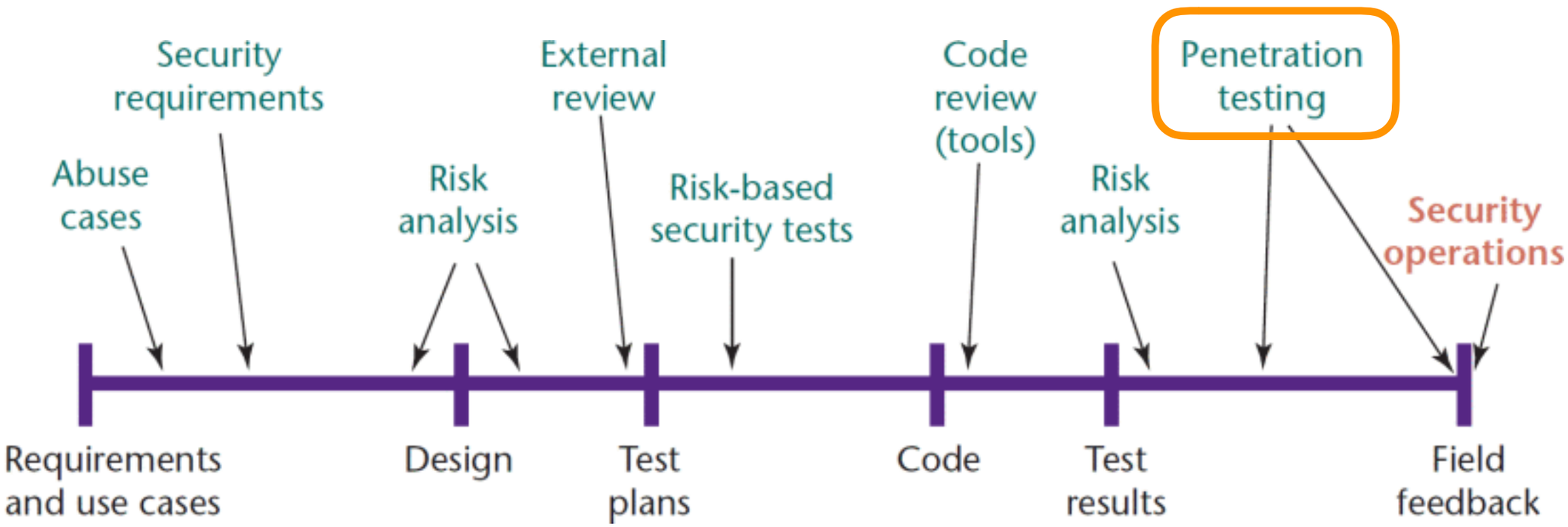
Risk-based security testing



- Security testing involves:
  - testing security mechanisms to ensure that their functionality is properly implemented, like other features
  - **but also** performing risk-based security testing motivated by understanding and simulating the attacker's approach, for instance in line with abuse cases …
  - more on this topic later in the course …

Pen Testing

Pen testing (penetration testing)



- **Pen testing**
  - Assess security by actively trying to find exploitable security vulnerabilities over a running system.
- Differs from standard software testing in that:
  - **Negative goal**: looks for unintended behavior rather than expected (unlike standard testing)
  - **Targets an entire system:** not applicable to individual modules (no unit testing)
  - **"black hat" activity**: conducted mostly by what are sometimes called "red teams", typically distinct from development or functional testing teams.
    - ✦ "Red teams" are pen-testing specialists, moreover standard developers & testers may also be naturally biased by their assumptions about the system.

Penetration testing tools

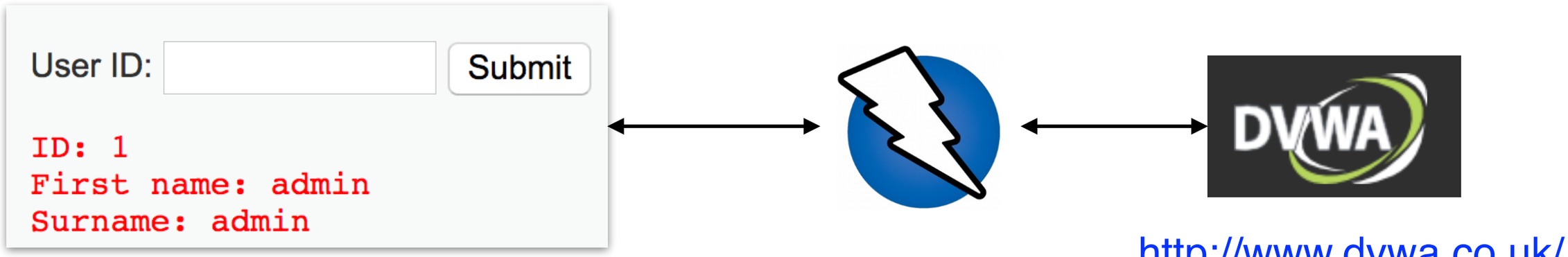
- Pen-testing tools run in automated fashion. Typical operation
  - **(1)** passively scan for vulnerabilities (“probing mode”),
  - **(2)** then try to see if they are exploitable (“attack mode”)
- Common aids:
  - “Libraries” of known exploits per type of vulnerability.
  - Fuzz testing techniques to derive malicious inputs.
- Some popular tools
  - Kali: a Linux distribution specifically aimed at pen-testing, including a vast amount of tools.
  - ZAP Web proxy
  - Metasploit framework
  - sqlmap for database pen-testing
  - nmap for network scanning

Example pen-testing tool — ZAP

Image source: [OWASP ZAP 2.7 Getting Started Guide]

- Probing mode
  - ZAP behaves as a HTTP proxy, sitting between the browser and a web application under tes
  - Potential aspects of vulnerability are detected & reported on-the-fly
- Attack mode
  - ZAP checks if potential vulnerabilities are exploitable
  - Pen-test results are reported

Pen-testing DVWA — example



http://www.dvwa.co.uk/

Probing mode

- X-Frame-Options Header Not Set
  - GET: http://localhost:8081/vulnerabilities/sqli/?id=1&Submit=Submit
- Web Browser XSS Protection Not Enabled
  - GET: http://localhost:8081/vulnerabilities/sqli/?id=1&Submit=Submit
- X-Content-Type-Options Header Missing
  - GET: http://localhost:8081/vulnerabilities/sqli/?id=1&Submit=Submit

Attack mode

- Cross Site Scripting (Reflected)
  - GET: http://localhost:8081/vulnerabilities/sqli/?id=%27%22%3Cscript%3Ealert%281%29%3E
- SQL Injection
  - GET: http://localhost:8081/vulnerabilities/sqli/?id=1%27+AND+%271%27%3D%271%27+-

Pen-testing and the SDLC

- **So**: pen-testing found a few issues that were fixed, great!
- **But wait**: how can you have have confidence that pen-testing is really being effective?
- McGraw warns:
  - *"people often use **penetration testing as an excuse to declare victory.** When a penetration test concentrates on finding and removing a small handful of bugs (and does so successfully), everyone looks good: the testers look smart for finding a problem, the builders look benevolent for acquiescing to the test, and the executives can check off the security box and get on with making money. Unfortunately, **penetration testing done without any basis in security risk analysis leads to this situation with alarming frequency.**"*
  - ***"security penetration testing can be effective, as long as we base the testing activities on the security findings discovered and tracked from the beginning of the software life cycle** […]. To do this, a penetration test must be structured according to perceived risk and offer some kind of metric relating risk measurement to the software's security posture at the time of the test. **Results are less likely to be misconstrued and used to declare pretend security victory."***
- ***Similar arguments can be made for other methodologies in the SDLC e.g. code reviews, risk-based security testing, …***