Introduction

Security Engineering

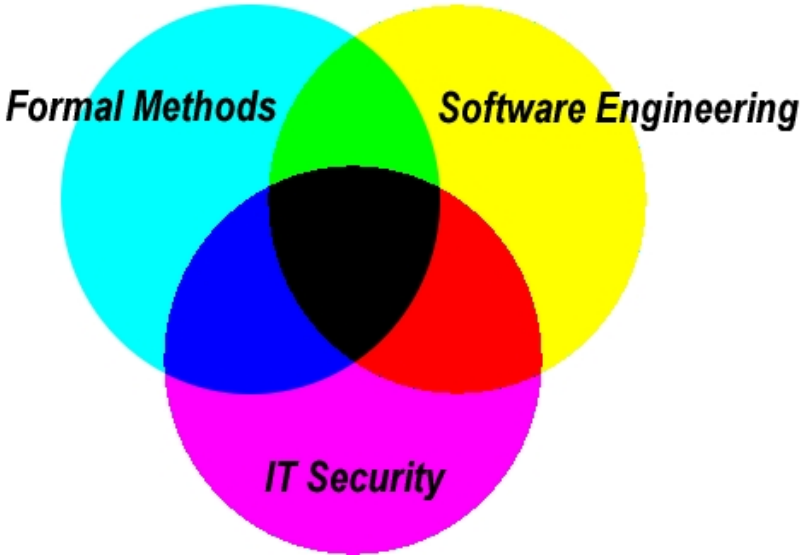
David Basin
ETH Zurich

Road map

☞ **Welcome and administrative details**

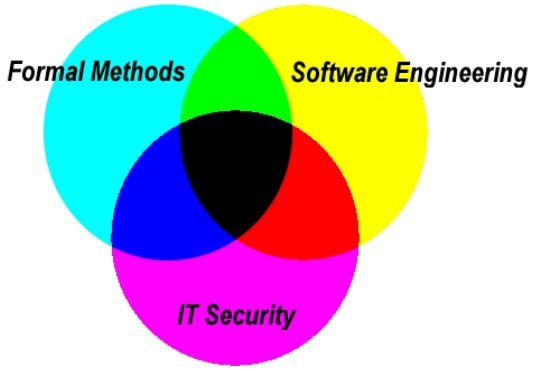
- Motivation and background
- Definitions
- Software engineering activities and where security fits in
- Contents and summary

Information Security Group @ ETH



- Offerings include
  - Security Engineering
  - Information Security Laboratory
  - Formal Methods for Information Security
  - Information Security Seminar
  - Numerous projects in Information Security
- More information at `www.infsec.ethz.ch`

Organization



- **Instructor:** David Basin
- **Assistants:** Marco Guarnieri, Lara Schmid
- **Prerequisites:**
  - Information Security course, or equivalent
  - Software Engineering course, or equivalent
  - Willingness to take initiative, read literature, **do assignments**
- **Format:** 2V+2U

  Lecture: Wed 10-12, CAB G.59, with 10 minute break
  Exercise: Wed 15-17, CAB G.51
- **Language:** English
  Please use English on assignments (grammar unimportant)

Homework, project, and final exam

- **Grade** determined by project and final (written) exam
  - Split: 20% project and 80% final exam
  - Further information in class and on web page

- **Homework** is not graded, but still **essential**!

  t_0**:** Exercises available on-line after lecture

  t_0**:** Brief discussion in exercise session (if necessary)

  $t_0 + 7$**:** Detailed discussion in exercise session

- **Exercises** begin this week, this afternoon

New and exciting

- **Latest results** and **development methods!**
- **Tool-based labs**
  - Model-driven development (used for project)
  - Code scanning and testing
- We hope this new material will excite you

  We will do some improvising along the way. Please bear with us
- Tool-specific detail and syntax introduced in labs

  Labs are an **essential** part of the course

What else?

- Slides, exercises, and other information on course web page
  http://www.infsec.ethz.ch/education/as2016/seceng

  We will try to have slides available before class

- Literature also listed there
  - No one book covers all lectures. Information widely scattered
  - Talk-specific references at end of most talks
  - Many resources available on web — try google!

- If you have questions, **please ask (language egal)**.

- Feedback of all kinds is welcome!

Road map

- Welcome and administrative details

☞ **Motivation and background**

- Software engineering activities and where security fits in

- Contents and summary

How security-critical systems are typically built

The result (example)

Sony Playstation Network Compromise[1]

- Account data for 77 million users stolen by network attacker
- Data included:

  Name, address, email, birthdate, Sony password/login, purchase history, billing address, credit card data, security question answer
- Direct costs ca. $175 million
- Loss of customers and reputation. Lawsuits.

IT-equivalent of theft of the crown jewels!

[1] Sources: IEEE Spectrum, April 28th, 2011 and NY Times June 10th, 2011

These are not singular incidents

Has Sony been hacked this week?

Time-line of Sony hacks (excerpt)[2]

2011-04-20 Sony PSN goes down

2011-05-21 Sony BMG Greece: data 8300 users **(SQL Injection)**

2011-05-23 Sony Japanese database leaked **(SQL Injection)**

2011-06-05 Sony Canada: 2,000 user credentials leaked **(SQL Injection)**

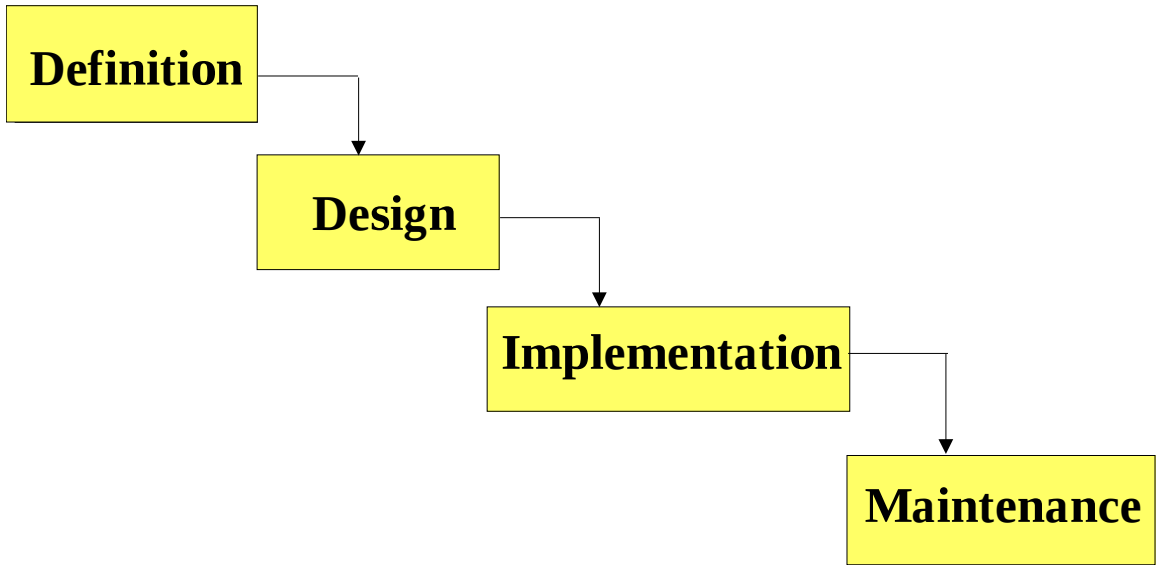
2011-06-05 Sony Pictures Russia **(SQL Injection)**

2011-06-06 Sony Portugal **(SQL injection, iFrame injection and XSS)**

2011-06-20 20th breach within 2 months
177k email addresses were leaked via a SQL injection

[2] Source: http://hassonybeenhackedthisweek.com

Why is the situation so pathetic?



Why is it so hard to build good software and even harder to build secure software?

Software is

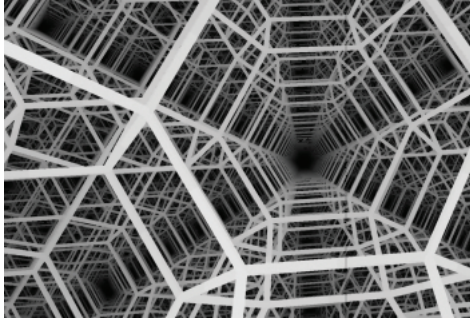
one of the most complex man-made artifacts

I believe the [spreadsheet product] I'm working on is far more complex than a 757 [jumbo jet airliner].

– Chris Peters, Microsoft

It's different [from other engineering disciplines] in that we take on novel tasks every time. The number of times [civil engineers] make mistakes is very small. And at first you think, what's wrong with us? It's because it's like we're building the first skyscraper every time.

– Bill Gates, Microsoft



Complexity

- **Lines of code**

Microsoft Word	approx. 1 million lines of code
Microsoft NT	approx. 16 million lines of code
Microsoft Vista	over 50 million lines of code

- Number of possible **system states** is a more accurate measure
  - an integer has 4.2 billion ($= 2^{32}$) possible values
  - an object with 2 ints and a boolean field has 40 thousand quadrillion values ($\approx$ grains of sand on all world's beaches)
  - How about Windows Vista?

One problem is just that of software engineering: scaling up methods to construct properly functioning, large-scale systems.

Complexity: the nature of the product

- Modern systems are **networked information systems** (NIS)
  - Integrate computer systems, communication systems, procedures, controllers, people, and even more
  - Information communicated includes software itself
- Characteristics of an NIS
  - Use **commercial off-the-shelf** (COTS) components
  - Use of semi-open architectures, e.g., closed system with third-party drivers or plugins
  - Cost pressures force quick, feature-oriented development
  - Code-base of 10^7-10^8 lines of code. Starting from scratch impossible.

Even with good development methods, defects abound[3]

- Industry average: 50-60 defects per 1000 lines of code prior to test
- Modern design methods reduce this to 20-50
- Final delivery has 2-4 defects per 1000 lines
- Code reviews can eliminate 50% of errors prior to test
- Verification-based reviews (cleanroom style) can eliminate 95%
- And yield 0.2 defects per 1000 lines at delivery[4] [5]

Is this acceptable? For what kinds of systems?

---

[3] Statistics: Myke Dyer, IBM Federal Systems Division Study, J. Software & Systems, 1987.

[4] Average reported in 2006 (CMU CyLab Sustainable Computing Consortium): 20-30 bugs per 1000 lines of code. In well-managed open-source projects, typically 0.4 - 1.22 bugs per 1000 lines.

[5] Compare: 2013 Coverity Scan Open Source Report: .59% for open source code and .72% for proprietary code.

Our focus: Security Engineering

What is so special about security?

Why is this problem particularly acute?

(1) Security is usually added on, not engineered in

- Standard security properties (CIA) concern **absence of abuse**

  Confidentiality (Secrecy): No improper disclosure of information

  Integrity: No improper modification of information

  Availability: No improper impairment of functionality/service

- Abuse-freeness means **restricting** system behavior. But...
  - It is more rewarding to **add** functionality to systems
    Programmers get quick feedback & managers get warm feelings
  - It appears easier to push security aside until later

- Typical result is **security as an after thought**

  Results in misanalyzed requirements, poorly engineered designs, inadequate verification and validation, etc.

(2) Software is not "continuous" or "almost right" is not good enough

- A minute change of 1 bit can have catastrophic consequences

  Security is the sum of countless little details and interactions

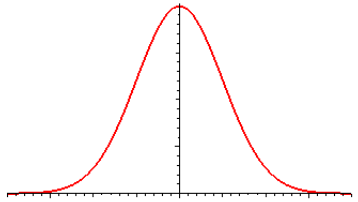
- It is precisely these minute failures that **hackers** exploit

  Hackers and their programs exploit the failed checks, incorrect settings, etc. that arise from minor oversights

- This is also a problem for functional correctness
  - But then just don't use the broken function (wait for patch)
  - And there is often more leeway, e.g., in developing systems for (soft) real-time or fault tolerant applications

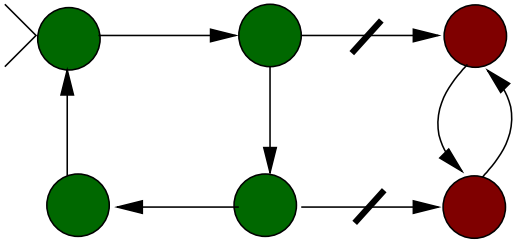
The hacker (adversary) plays a key role here

(3) Hackers are not typical users



- Systems should work correctly in intended operating environment
  - **Example:** Microsoft Word should function reliably for typical documents processed by typical users
  - **Example:** Buildings must be stable for an expected range of usage, weather conditions, etc.

- A system is **safe** (or **secure**) if the environment cannot cause it to enter an unsafe (insecure) state



So, abstractly, security is a **reachability** problem.
What states can the environment drive the system to?

Hackers are not typical users (cont.)

- The environment includes the hackers
  - They are malicious! Their actions are neither typical nor random
  - Their raison d'être is to force the system into an insecure state

- In a world of **saints** we could focus our energies elsewhere, but on planet earth we must be prepared for **devils**

  Surveys showed that the great majority of security failures resulted from the opportunistic exploitation of various design and management blunders. In most system engineering work, we assume that we have a computer [more generally, an environment] which is more-or-less good and a program which is probably fairly bad. However, it is helpful to consider the case where the computer [environment] is thoroughly wicked. In other words, the black art of programming Satan's computer might give us insights into the more commonplace task of trying to program Murphy's. [6]

[6] Ross Anderson and Roger Needham, Springer LNCS 1000.

An example — race conditions

- Certain operations are composed of discrete steps and it is possible for attackers to take actions in-between them
- An example: if lock l not present then create l and ...

  Here l could be a lock file, a bit set to 1 in shared memory, etc.
- A more concrete example (source: FreeBSD advisory SA-02:08)

  A race condition exists in the FreeBSD exec system call implementation. A user can attach a debugger to a process while it is exec'ing, but before the kernel has determined that the process is set-user-ID or set-group-ID. Local users may thereby gain increased privileges on the local system.

  All versions of FreeBSD 4.x prior to FreeBSD 4.5-RELEASE are vulnerable to this problem. The problem has been corrected by marking processes that have started but not yet completed exec with an 'in-exec' state. Attempts to debug a process in the in-exec state will fail.

4) The adversary can exploit not only the system but also the world

Key point: Security requirements concern system's effect on the world!

What is Security Engineering?

- **A rough definition**

  Security Engineering = Software Engineering + Information Security

- **Software Engineering** is the application of systematic, quantifiable approaches to the development, operation, and maintenance of software; i.e., applying engineering to software.

- **Information Security** focuses on methods and technologies to reduce risks to information assets.

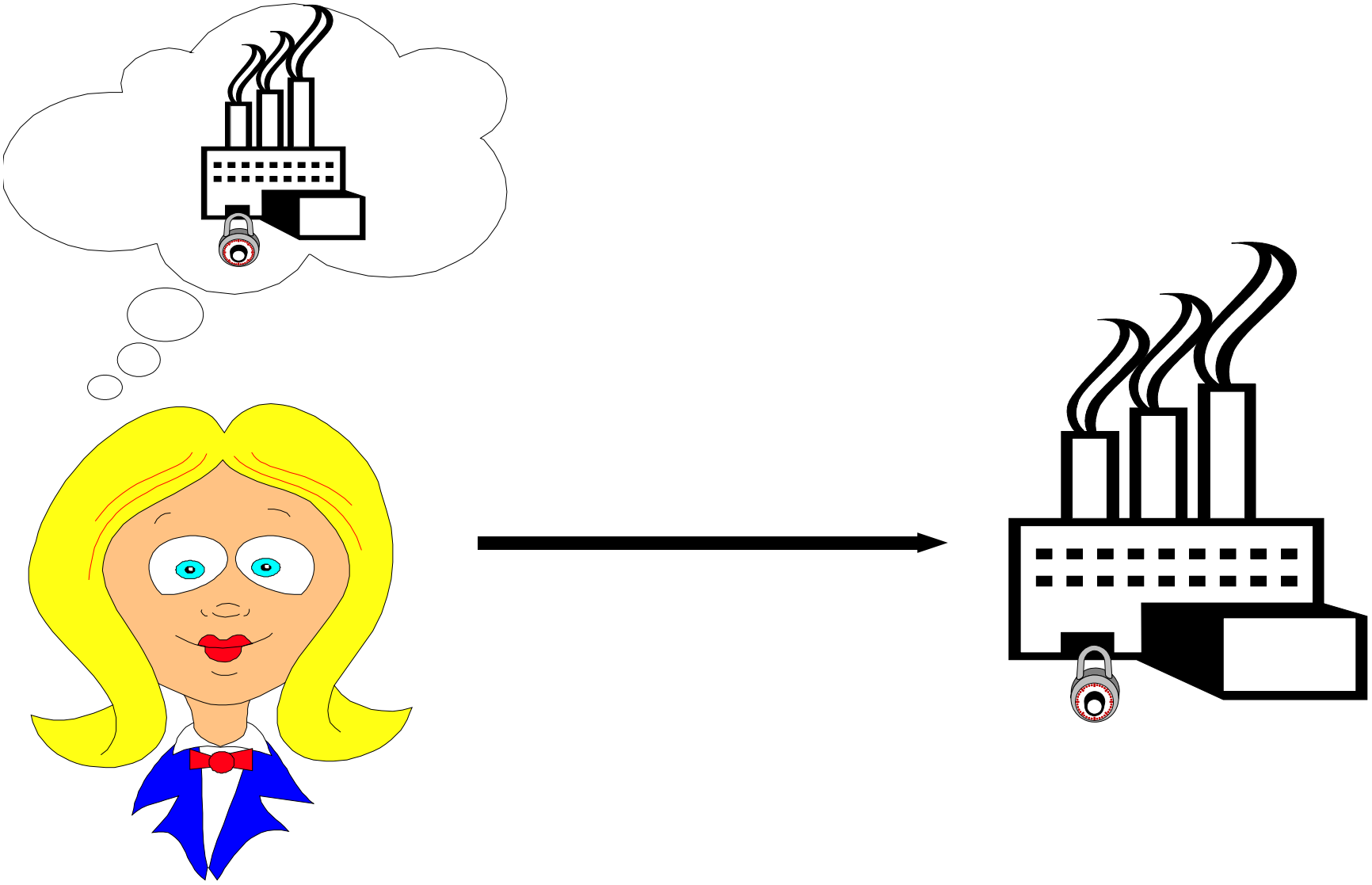
- **A more refined definition**

  Security Engineering concerns building systems that operate securely in the face of error, mischance, and, in particular, malice. As a discipline, it focuses on the tools, processes, and methods to design, implement, test, and maintain systems.

Road map

- Welcome and administrative details
- Motivation and background
- ☞ **Software engineering activities and where security fits in**
  - Focus on **process models**
  - How to organize and manage the construction of (secure) systems
- Contents and summary

The big picture



Software development — historically

- The **code-and-fix** development process

  1. Write program
  2. Improve it (debug, add functionality, improve efficiency, ...)
  3. GOTO 1

- Acceptable for 1-man projects and first-year CS assignments

- Problematic for larger projects. Reasons include:
  - Not transparent. No checkpoints
  - Disastrous when programmer quits
  - Expectations often differ when the developer isn't the end-user
  - Maintainability? Security?

Code-and-fix variant: penetrate-and-patch

1. System is released

   What once was **code/debug/release** is now **release/debug/code**

2. Bad guys analyze it to death (beta-testing for free)

3. Vulnerabilities discovered that are afterwards (pick one):

   - disclosed to system developers or bug-bounty programs
   - turned into exploit code, downloaded to worms, and released on the world
   - used by nation states for espionage and cyber warfare

4. Race to develop, distribute, and install patch

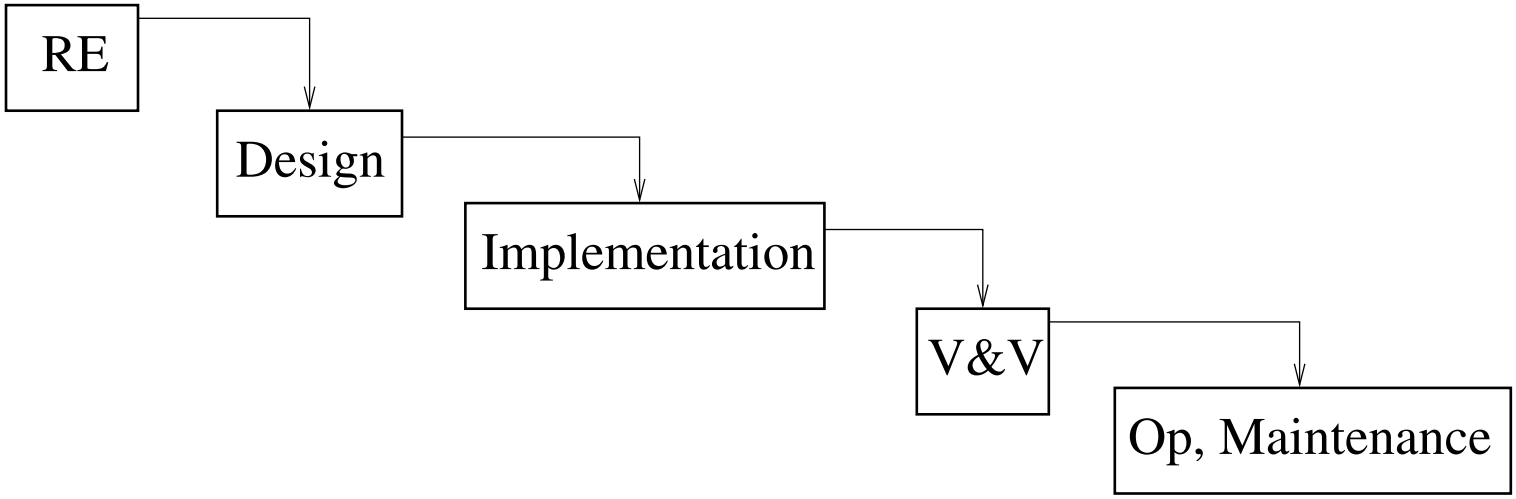
5. GOTO 2

Alternatives?

- Many **development processes** have been proposed to structure **development activities** and their **associated results**
- Goal is to improve the quality and maintainability of the resulting systems and decrease production costs
- Let's look at the classic example: the **waterfall model**

  Not just historically interesting: phases found in most alternatives
- Here we examine relevant **software-engineering** activities
  - Later we cover **security-relevant** subactivities
  - Now we just mention connection points

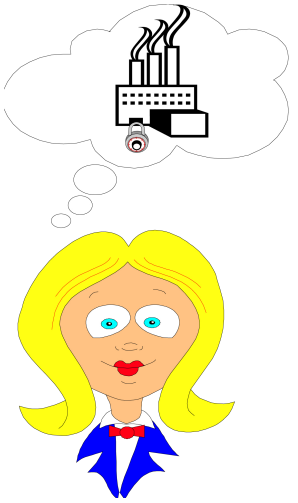
Waterfall model (Royce 1970)



- First process model (also called **phase model**)
  - The development is decomposed into phases
  - Each phase is completed before the next starts
  - Each phase produces a product (document or program)
- Enthusiastically welcomed by managers!
- Now called the **Systems Development Life-Cycle** (SDLC)

Requirements engineering

What should the system do?



- **Objective:** analyze and document system requirements
- **Build models** to work out requirements and make them precise
- Functional system properties

  E.g., compute Fourier transform, sort database table, ...
- Non-functional system properties
  - **Security**, dependability, performance, usability, ...
  - Often system-wide properties.
  - Also related to interaction with, and effects on, the world.
- Related activities: analyze risks, determine priorities, study feasibility (e.g., via simple prototypes), make business case

Design
How to do it (abstract)?

- Requirements decomposed into soft- and hardware requirements
  - Fix system-architecture, with requirements for each subsystem
  - Specify (sub)system interfaces
  - Define relationships between systems, e.g., message flows, synchronization, etc.
- Design of subsystems (recursive decomposition)
- Design of main data structures and algorithms
- Various structured design methods employed, e.g., UML
- **For security**: map requirements to security technologies

  Encryption, access control, key-management, logging, ...

Implementation
How to do it (concrete)?

- Develop programs for different subsystems

  Includes further design, e.g., data structures and algorithms

- Often driven by using existing code base or libraries
  - Reuse speeds development and may increase reliability
  - Or alternatively propagate errors

- **For security**
  - Programmers must understand security implications of their code
  - Use "secure languages", follow secure coding guidelines, use analysis tools, code reviews, etc.

Validation and verification

Did we get it right?

- Program inspection: line-by-line review by team
- Static analysis: automated program inspection.
  For security: focus on vulnerability detection
- Theorem proving: interactive program verification
- Testing
  - Black-box (specification based) and white-box (code based)
  - Regression testing against previous versions
  - Testing at different levels: unit, module, integration, system, and acceptance testing
  - **For security**: risk-based testing and penetration testing

How do objectives/guarantees of these activities differ?

Operation and maintenance

- Installation, patches, improvements, technology upgrades, ...
- Seems boring and straightforward. But it isn't!
- Absolutely critical from security standpoint. E.g.,
  - Secure distribution of source and updates/patches
  - Configuration and setup, e.g., access control, key distribution
  - Bug tracking and fixing on development side
  - Backup, failure recovery, continuity planning
  - User education on (securely) using software. Help desk support.
- Support for these tasks is crucial for project success

  Requires careful planning and often considerable resources

Advantage: a clearly structured, transparent process

Activity	Result
Requirements analysis	Feasibility study, requirements sketch
Requirements definition	Requirements plan
System specification	Functional specification
	Test plan,
	Development of user documentation
Architecture development	Architecture specification
	System test plan
Interface development	Interface specification
	Integration test plan
Detailed development	Specification, unit test plan
Programming	Program
Unit test	Report
Module test	Report
Integration test	Report, final user documentation
System test	Report
Acceptance test	Report, documentation

The output of one phase is the input to the next

However, waterfall makes strong assumptions

- Requirements are known from the start, before design
- Requirements rarely change
- Design can be conducted in a purely abstract way
- Everything will all fit nicely together when the time comes

Problems with the waterfall

- The assumptions are too strong!

  E.g., requirements usually imprecise and mature during development

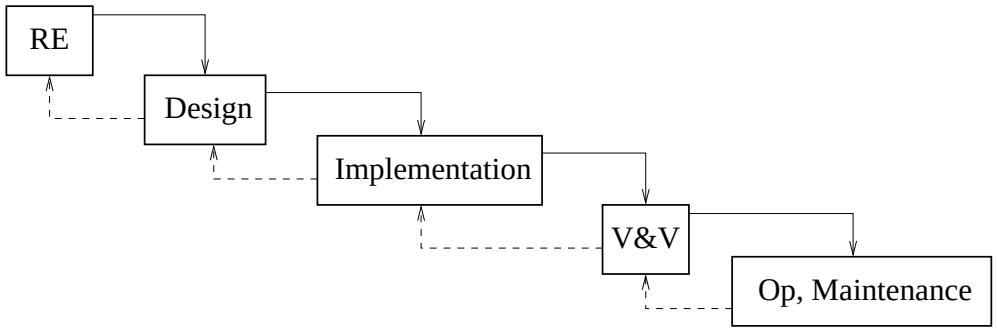
- **Big Bang Delivery Theory** is risky: proof of concept only at end!

- Too much documentation! (Paper flood $\Rightarrow$ CASE tools)

- Late deployment hides many risks
  - Technological (well, I **thought** they would work together...)
  - Conceptual (well, I **thought** that's what they wanted ...)
  - Personnel (took so long, half the team left)
  - Users **see** nothing real until the end, and they always **hate** it!

- Testing comes in too late in the process

Problems (cont.)



- Unidirectional flow too stiff

  Problems are pushed to others or programmed around

- Alternative: allow arbitrary feedback (the backwards arrows)

  New problem: iteration makes checkpoints difficult

- Other alternatives to structuring development.
  - Variants depending on organization, country, and product
  - Examples: V-model, RUP, and agile methods
  - For other models: see references at end

V-Model

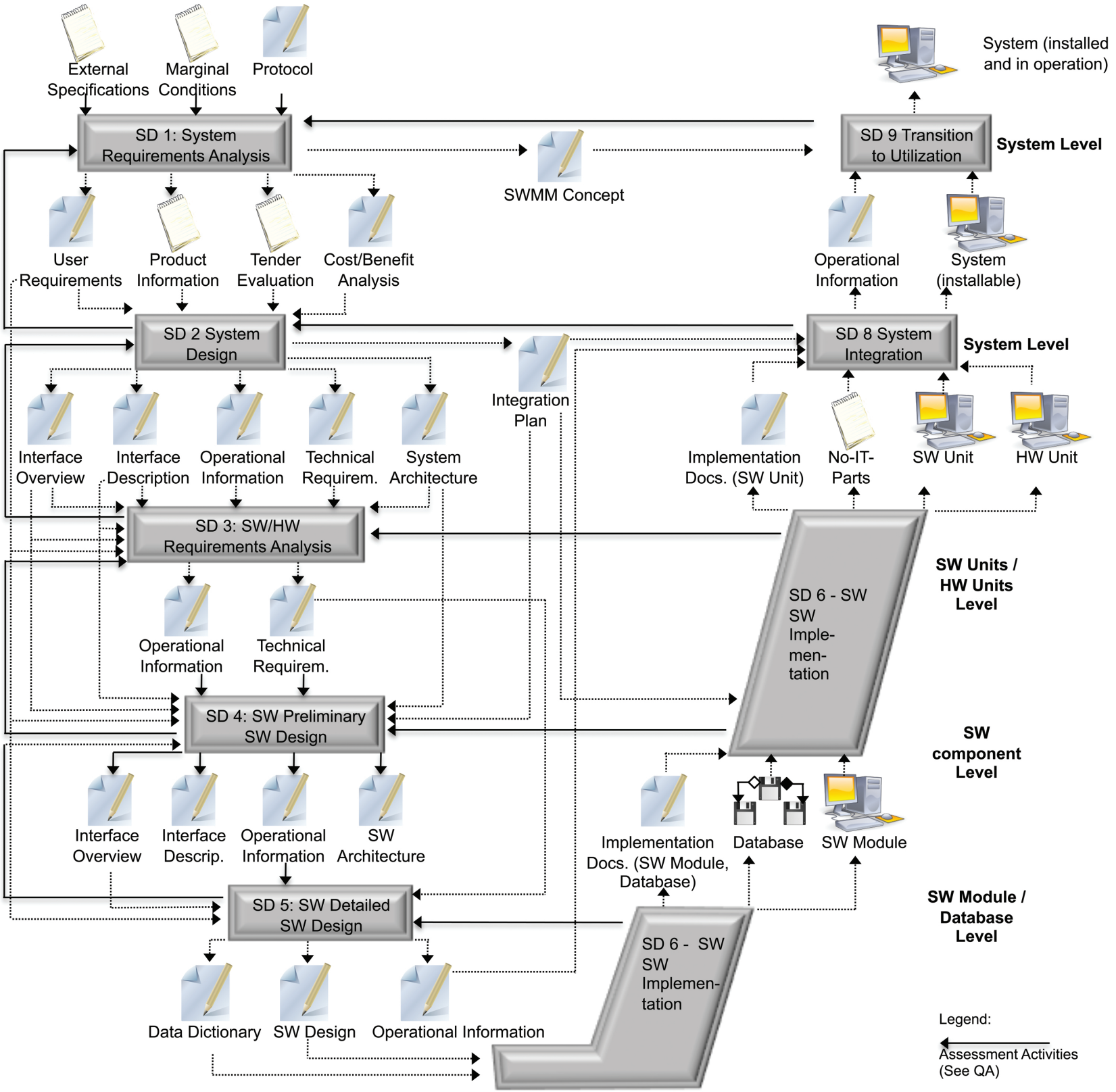
- A model for military and administrative projects in Germany

  ISO standard: regulates all **activities**, **products**, and their **states** and **relationships** during IT-development and maintenance

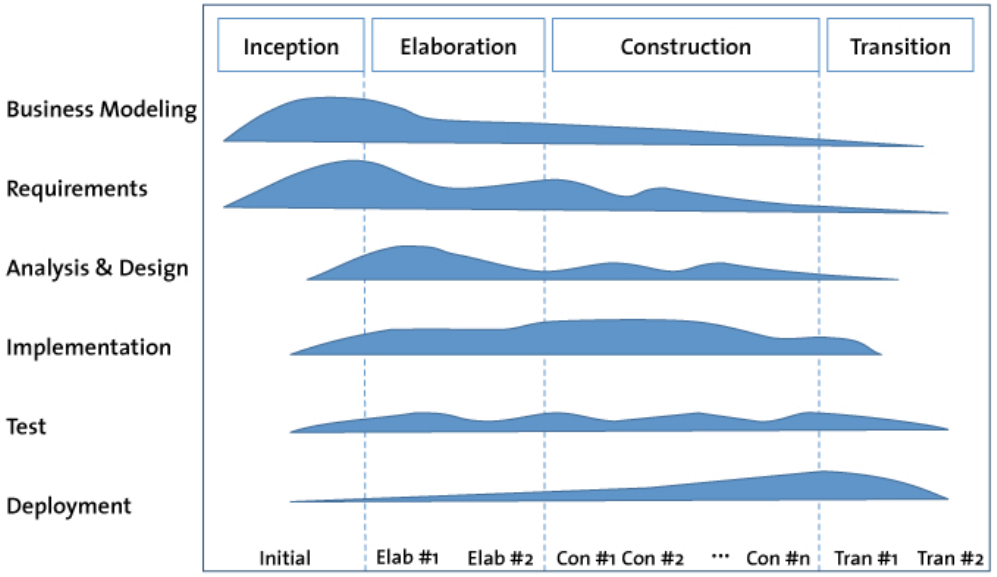
- Built from different submodels, including:

  system development, configuration management, project management (purchasing, planning, ...), etc.

- System development model can be viewed as a heavyweight V-formed variant of the waterfall


External Specifications
Marginal Conditions
Protocol
SD 1: System Requirements Analysis
SWMM Concept
SD 9 Transition to Utilization
System (installed and in operation)
System Level
User Requirements
Product Information
Tender Evaluation
Cost/Benefit Analysis
Operational Information
System (installable)
SD 2 System Design
SD 8 System Integration
System Level
Integration Plan
Interface Overview
Interface Description
Operational Information
Technical Requirem.
System Architecture
Implementation Docs. (SW Unit)
No-IT-Parts
SW Unit
HW Unit
SD 3: SW/HW Requirements Analysis
SD 6 - SW SW Imple-men-tation
SW Units / HW Units Level
Operational Information
Technical Requirem.
SD 4: SW Preliminary SW Design
SW component Level
Interface Overview
Interface Descrip.
Operational Information
SW Architecture
Implementation Docs. (SW Module, Database)
Database
SW Module
SD 5: SW Detailed SW Design
SW Module / Database Level
SD 6 - SW SW Implemen-tation
Data Dictionary
SW Design
Operational Information
Legend:
Assessment Activities (See QA)

Rational Unified Process



- Iterative waterfall variant championed by IBM/Rational
- Aims at minimizing risks

  Plan a little, design a little, code a little
- Phases

  Inception: Rough system definition for initial costing

  Includes: basic use cases, project plan, initial analysis of **business** risks

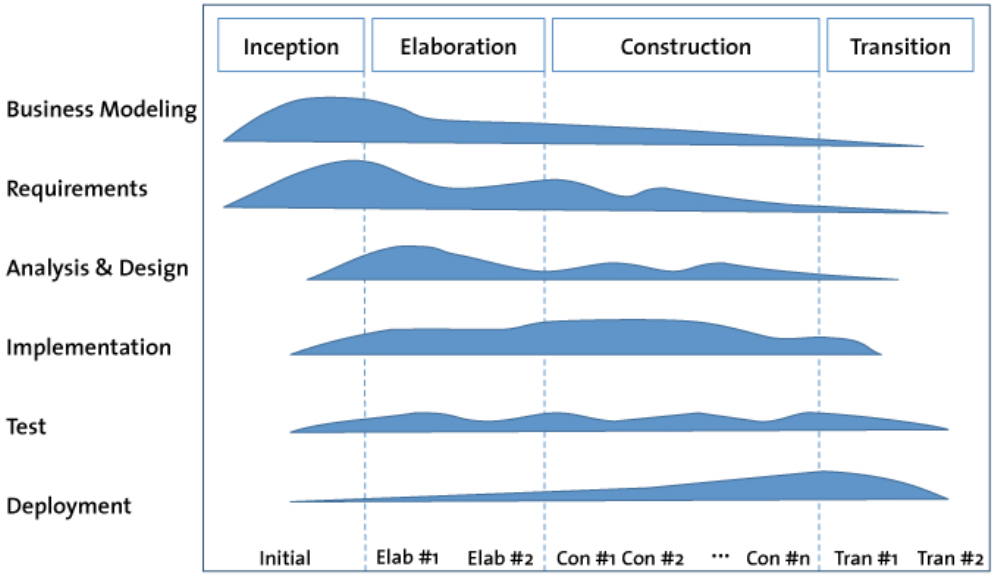
  Elaboration: Mitigate key risk items or redesign/cancel project

  Includes: elaboration of use cases, software architecture, executable artifacts for most risky parts, and development plan for overall project

  Construction: build system, elaborate features, first release

  Transition: improve system, beta test, validate, train users

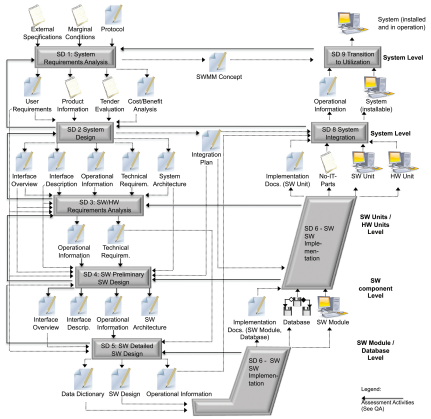
Rational Unified Process



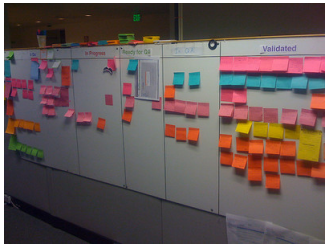
- Advantages
  - Reduce risk of failure by breaking system into mini-projects, focusing on riskier elements first
  - Encourages all participants, including testers, integrators, and documenters, to be involved earlier on
  - Mini-waterfalls centered around UML & OO-development
- Does it work?

  Experience seems positive although benefits difficult to quantify

Agile Methods

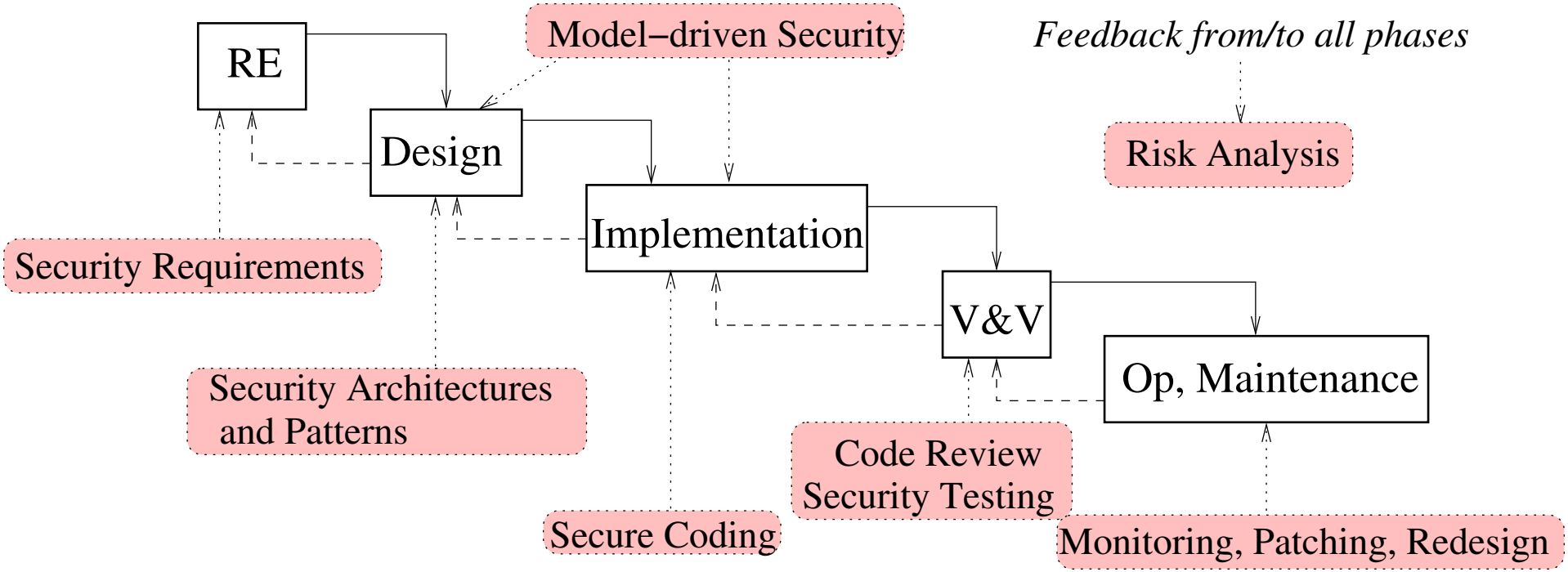


- Previous methods can be **heavyweight**, requiring considerable documentation and tool support
- Alternative **lightweight** agile methods put software first
  - Team based: small groups of 2-7 members
  - Tasks decomposed into small increments where team works through all phases
  - Regular meetings and face-to-face communication. Documentation is de-emphasized.



  - Regular interaction with customer, e.g., concerning requirements and whether project optimizes return-on-investment
- Numerous variants: Scrum, Extreme Programming, ...
  All are midpoints between code-and-fix and waterfall/V model

So where does security fit in?



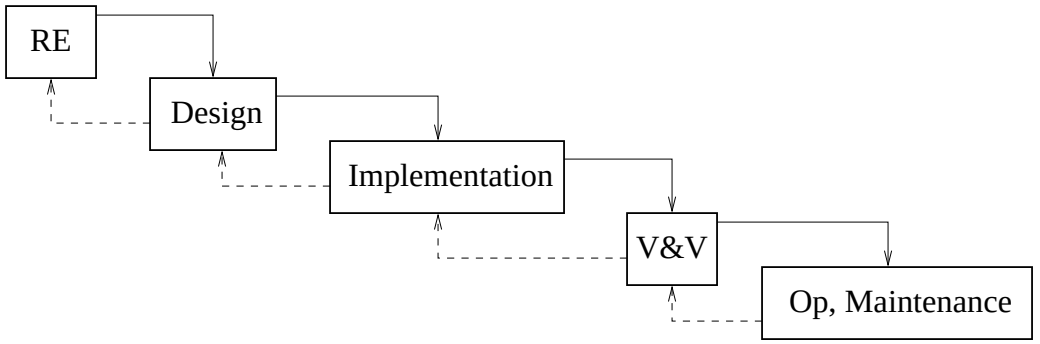
No standard development processes explicitly support security.
Security usually treated as an ad-hoc add-on at the end.

Road map

- Welcome and administrative details
- Motivation and background
- Software engineering activities and where security fits in

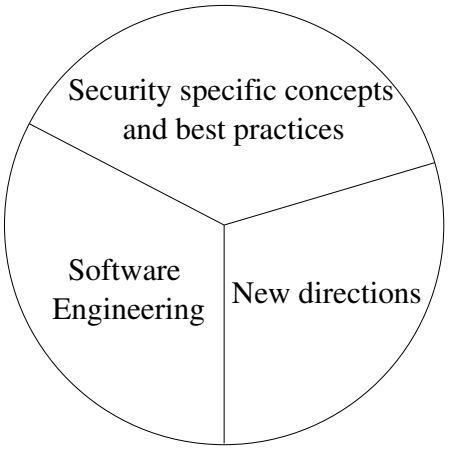
☞ **Contents and summary**

Topics covered



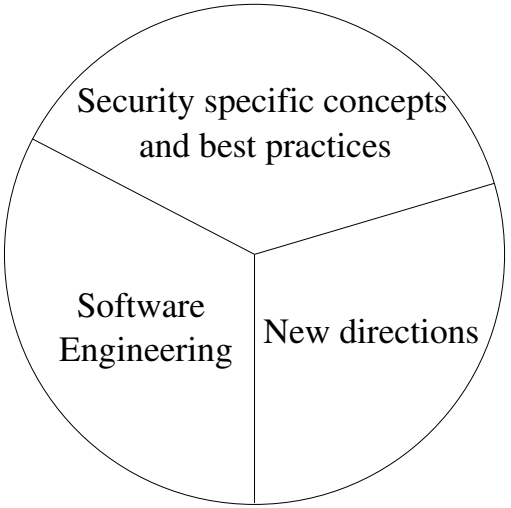
- Modeling foundations
- Security requirements analysis
- Security policy modeling and model driven development
- Security design principles
- Implementation-level security
- Risk analysis and system evaluation
- Code review, static analysis, testing

Methodological focus and learning objectives



- Techniques, methods, and tools for building secure systems
  - Large overlap with traditional Software Engineering
  - but emphasis on security aspects
- Security by construction rather than penetrate and patch
- Risk analysis and reduction methods
- An understanding both of the status quo and better alternatives
- **NOT** latest vulnerability, attack, network appliance, ...

Focus/objectives (cont.)



- **Models** will play a central role

  A model is a representation of a system that clarifies the system's structure and properties

- Reasons for using models
  - Abstraction and generality: can focus on key features
  - Visualization, documentation, comprehensibility
  - Decisions are explicit and precise
  - Analysis possible

- We will use models primarily based on UML

Summary — lessons learned

- Methods and tools are needed to master the complexity of software production
- Security needs particular attention
  - security aspects are typically poorly engineered
  - systems usually operate in highly malicious environments
- One needs a structured development process (discussed here)

  with specific support for security (coming lectures)

Literature

- Ross Anderson, Security Engineering, Wiley, 2001. (Parts online.)
- Trust in Cyberspace, Committee on Information Systems Trustworthiness, National Research Council (1999).
- W. W. Royce, *Managing the development of large software systems: concepts and techniques*, ICSE '87: Proceedings of the 9th international conference on Software Engineering, 1987.
- Barry W. Boehm, *A Spiral Model of Software Development and Enhancement*, Computer, 1988.
- Watts S. Humphrey, *Managing the Software Process*, 1989.
- The Unified Software Development Process, I Jacobson, G Booch, J Rumbaugh, Addison-Wesley, 1999.
- V-Model: Google "V-Modell".