Software Engineering

Development Processes

Peter Müller

Chair of Programming Methodology

The slides in this section are partly based on the courses
“Software Engineering I” by Prof. Bernd Brügge, TU München and
“Software Engineering” by Prof. Jan Vitek, Purdue University

Spring Semester 10

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

8. Development Processes

8.1 Classical Process Models

8.2 Extreme Programming

Requirements for Development Process

- A procedure to guide and control the entire development
- Should support developing high-quality systems
  - Software qualities: see Lecture 1
  - Acceptable development costs (time, money, etc.)

Typical Development Process Questions

- **Which activities** to select for the software project?

- What are the **dependencies** between activities?
  - Does system design depend on analysis?
  - Does analysis depend on design?

- How to **schedule** the activities?
  - Should analysis precede design?
  - Can analysis and design be done in parallel?
  - Should they be done iteratively?

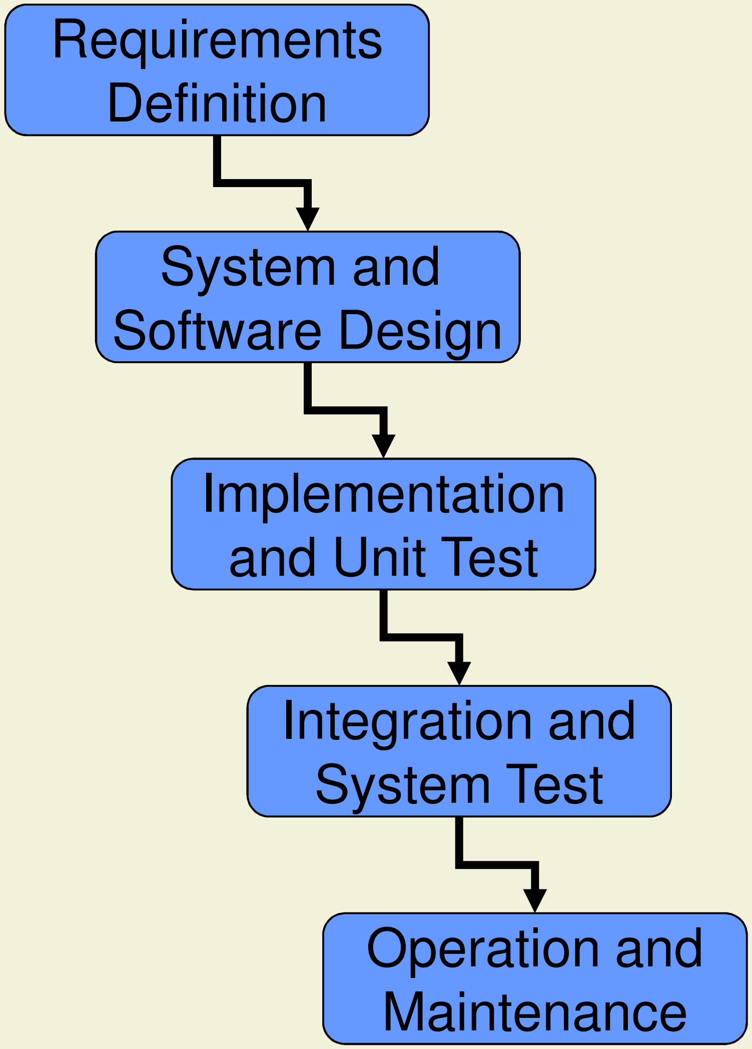
Software Development Activities

- Development process: **activities and results** of software production
- Four basic activities:
  - **Specification**: Definition of functionality and constraints
  - **Development** and Implementation: Production of system
  - **Validation**: Verification, testing, etc.
  - **Maintenance**: Changes and improvements
- Subdivision of activities depends on the particular process employed
- Differs depending on the kind of system built and the organizational context

Software Development Activities (cont’d)

Problem Domain	Requirements Analysis	What is the problem?
	System Design	What is the solution?
Implementation Domain	Detailed Design	What are the mechanisms that best implement the solution?
	Implementation	How is the solution constructed?
	Testing	Is the problem solved?
	Deployment	Can the customer use the solution?
	Maintenance	Are enhancements needed?

Waterfall Model (Royce 1970)


Requirements Definition
System and Software Design
Implementation and Unit Test
Integration and System Test
Operation and Maintenance

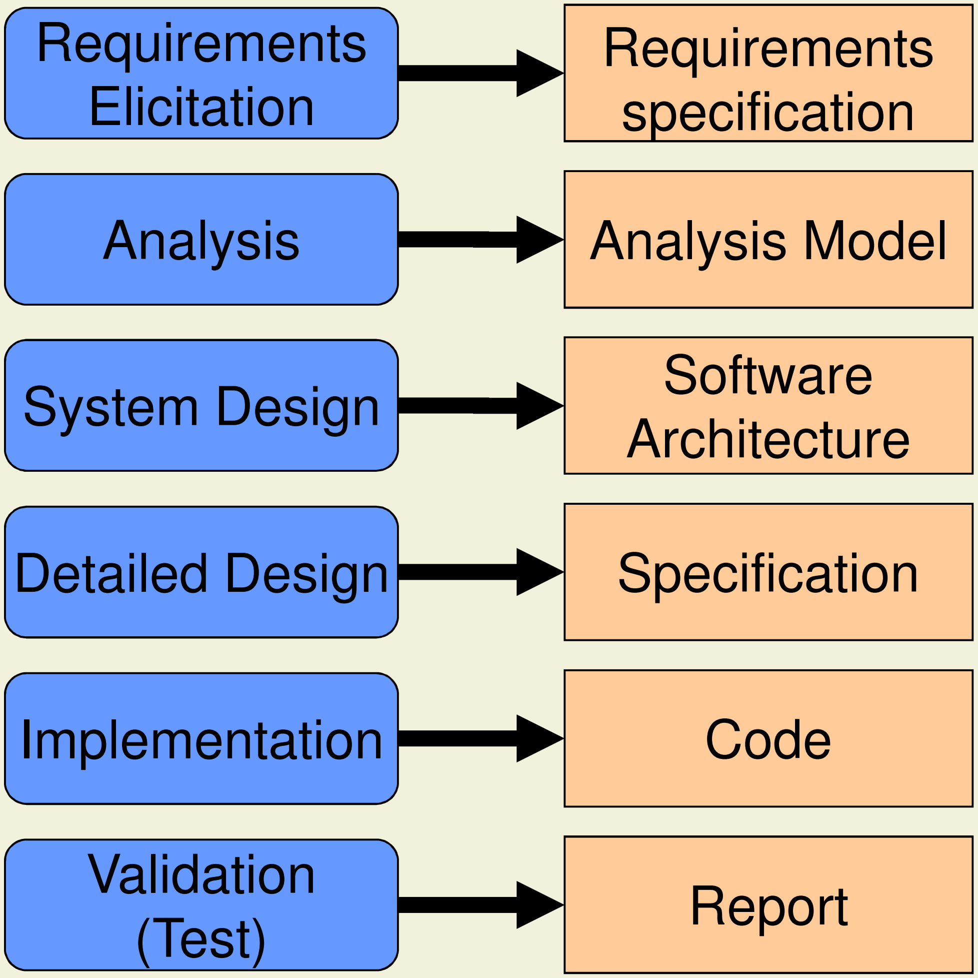
- First process model (also called phase model)
  - The development is decomposed in phases
  - Each phase is completed before the next starts
  - Each phase produces a product (document or program)
- Loved by managers
  - Nice milestones
  - Easy to check progress

Waterfall Process 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

Advantage of Waterfall: Transparency

- The output of one phase is the input of the next



Requirements Elicitation → Requirements specification

Analysis → Analysis Model

System Design → Software Architecture

Detailed Design → Specification

Implementation → Code

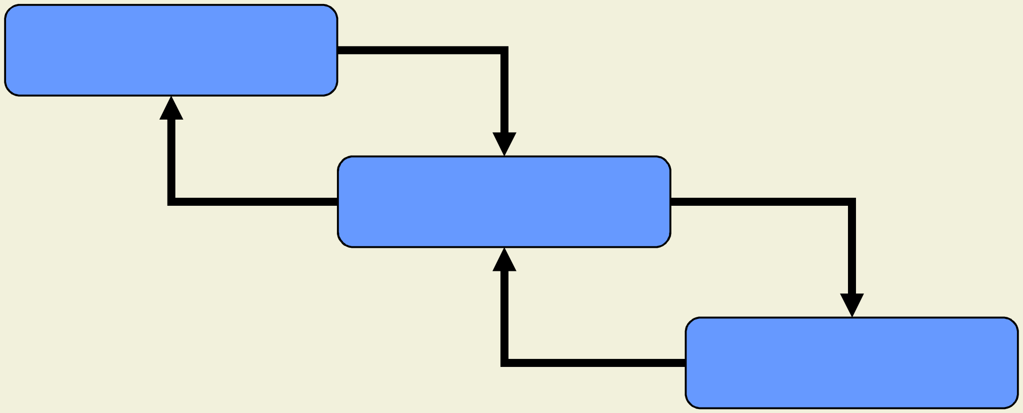
Validation (Test) → Report

Problems with the Waterfall

- **Assumptions** typically **don’t apply**
  - E.g., requirements typical imprecise and mature as development advances
  - Big Bang Delivery risky: proof of concept only at the end
- Late deployment **hides many risks**
  - Technological (“I thought they would work together...”)
  - Conceptual (“I thought that’s what they wanted ...”)
  - Personnel (took so long, half the team left)
  - User doesn’t see anything real until the end, and they always hate it
  - Testing comes in too late in the process

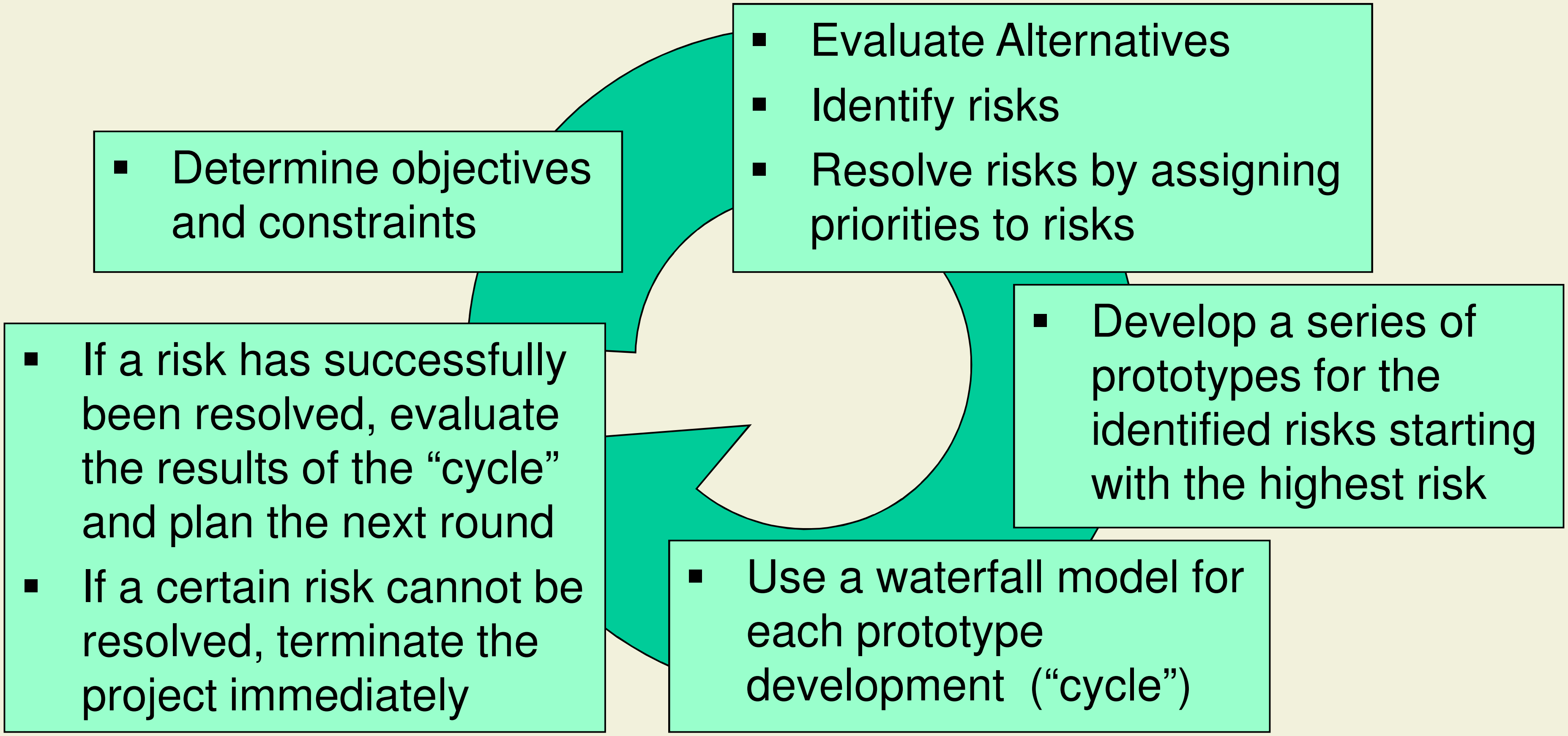
Problems with the Waterfall (cont'd)

- Too much documentation (paper flood)
- Unidirectional flow often too stiff: **feedback** is needed between phases
  - Design reveals problems in requirements
  - Coding reveals design and requirement problems, etc.
- Alternative: weakening through feedback



Spiral Model (Boehm 1985)

- Idea: build a prototype and continually improve it
  - Build in customer feedback at each iteration



- Determine objectives and constraints

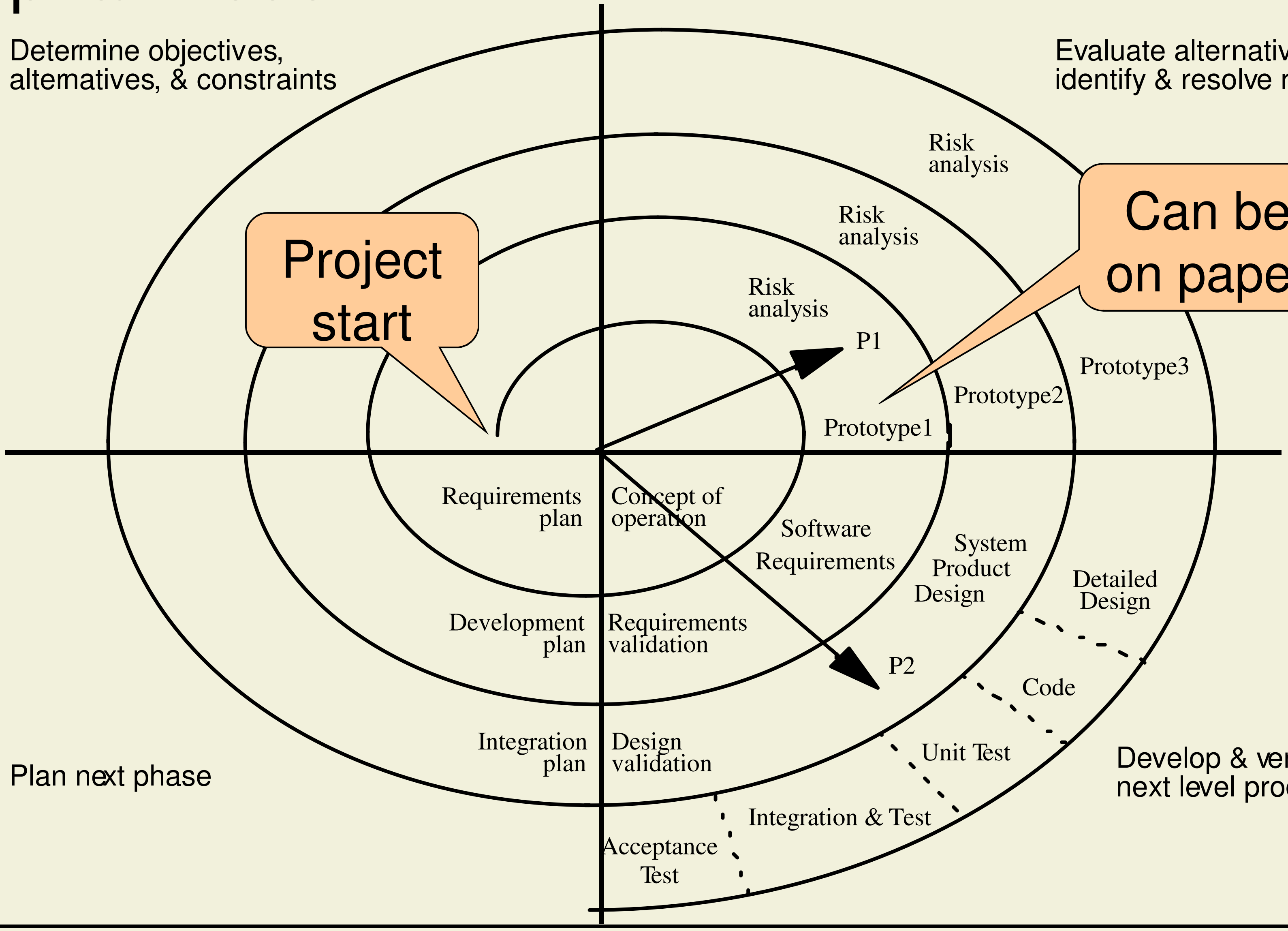
- Evaluate Alternatives
- Identify risks
- Resolve risks by assigning priorities to risks

- Develop a series of prototypes for the identified risks starting with the highest risk

- Use a waterfall model for each prototype development (“cycle”)

- If a risk has successfully been resolved, evaluate the results of the “cycle” and plan the next round
- If a certain risk cannot be resolved, terminate the project immediately

Spiral Model



Types of Prototypes

- Revolutionary Prototyping
    - Get user experience with a throwaway version to get the requirements right, then build the whole system
- Evolutionary Prototyping
    - The prototype is used as the basis for the implementation of the final system
    - Advantage: Short time to market
    - Disadvantage: Can be used only if target system can be constructed in prototyping language

Spiral Model: Discussion

- Theoretically, wide applicability
    - Many systems built this way
- Problematic
    - Not transparent: Difficult to judge progress. Managers have no checkpoints
    - Poorly structured code: due to frequent modification
    - Requires a skilled team: Small, skilled, and motivated group
- Practically, narrow applicability: small systems with limited life times

8. Development Processes

8.1 Classical Process Models

8.2 Extreme Programming

Extreme Programming (XP)

- A light-weight methodology for small to medium sized teams
- Developing software in the face of vague and **rapidly changing requirements**

- XP and traditional methodologies
  - XP runs counter to software engineering practice
  - XP is not a solution for all problems
  - XP is programmer friendly

- Extreme Programming is an **Agile Method**

Problems Addressed by XP

- Schedule slips
  - Delivery date is always six months in the future
- Project canceled
  - After many slips, project canned
- System goes sour
  - After a couple of years of operation and some changes, bugs start to appear
- Defect rate
  - So buggy that it is not used
- Business misunderstood
  - Software does not answer all the right questions
- Business changes
  - System answers the wrong (out of date) questions
- False feature rich
  - Lots of unused features
- Staff turnover
  - Where have all the good programmers gone?

XP Approach

- Schedule slips
  - **Short release cycles** to limit the scope of slips
  - Within release, 1 to 4 weeks customer-requested feature iterations
  - Within iteration, 1-3 day tasks
  - Implement most important features first, to minimize the impact of slips
- Project canceled
  - **Customer involvement** to choose the smallest possible release, to minimize potential bottlenecks and maximize software value

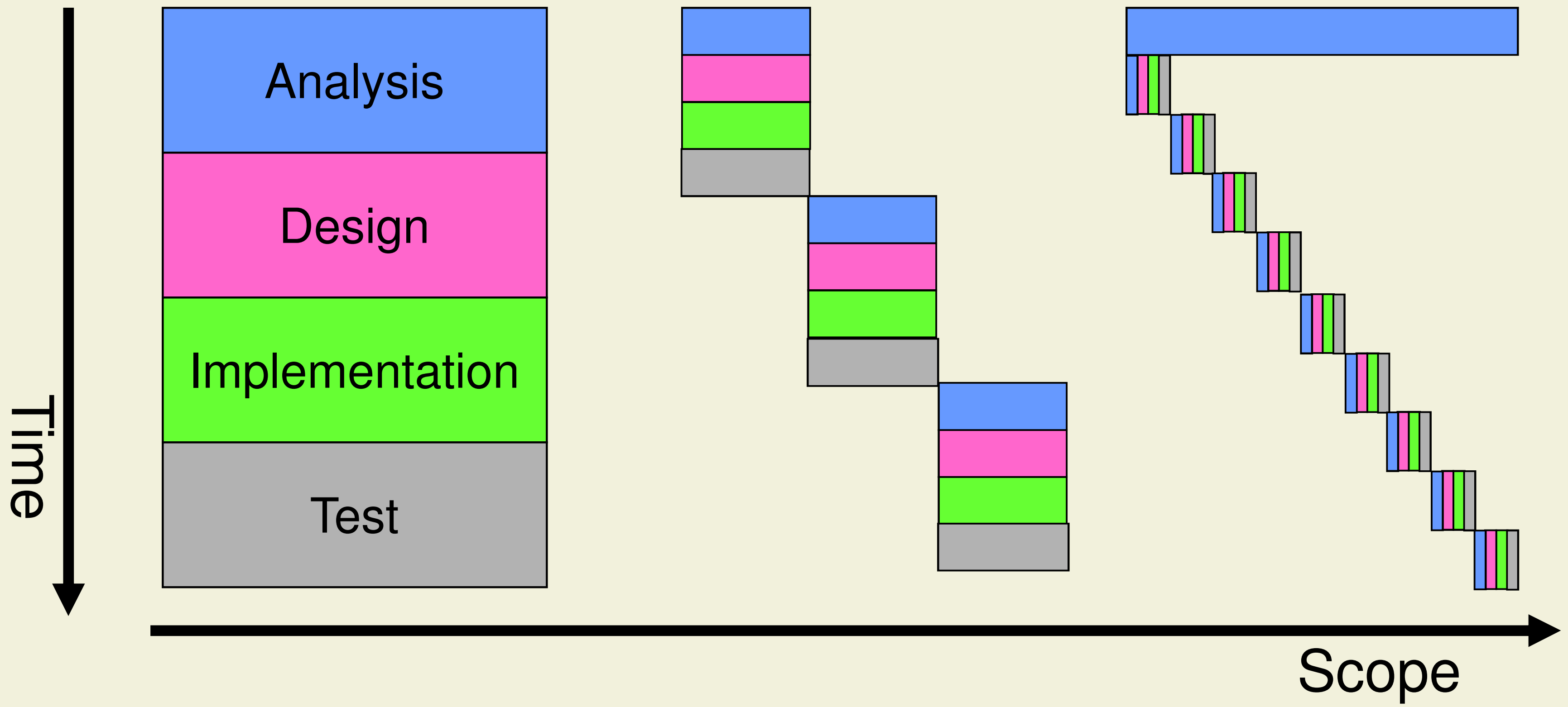
XP Approach (cont’d)

- System goes sour
  - Create and maintain a **comprehensive suite of tests**
  - Run tests **after every change**
- Defect rate
  - Unit test (programmer defined)
  - Functional tests (user defined)
- Business misunderstood
  - **Customer** is an integral **part of the team**
  - Specification **continuously refined**

XP Approach (cont'd)

- Business changes
  - Shorter release cycles imply less change during development
  - Unimplemented features can be replaced at no cost
- False feature rich
  - Only highest priority tasks are addressed
- Staff turnover
  - Religion
  - Shared code ownership

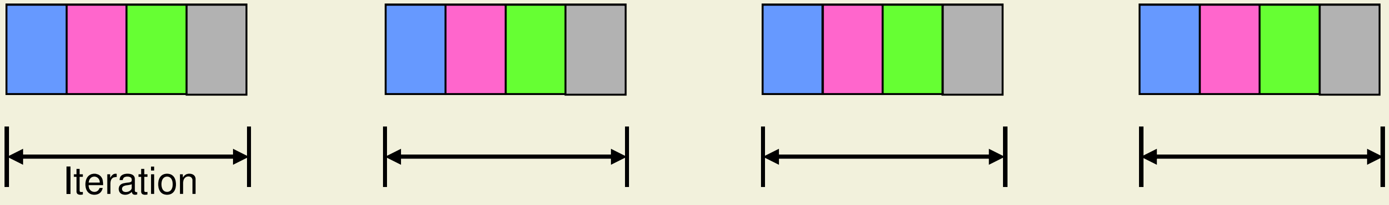
Extreme Programming



- Suggested reading: Kent Beck: *Embracing Change with Extreme Programming*, 1999

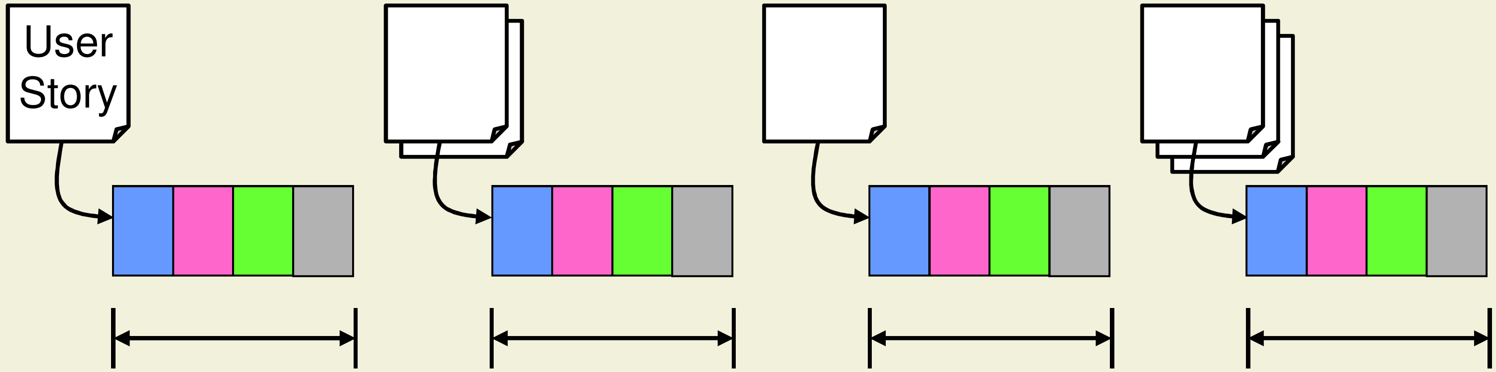
Extreme Programming: Iterations

- Typical **duration** between **2** and **4 weeks**
- **Timeboxed**: Iteration finishes always on fixed date
- Number of implemented **features** is **variable**



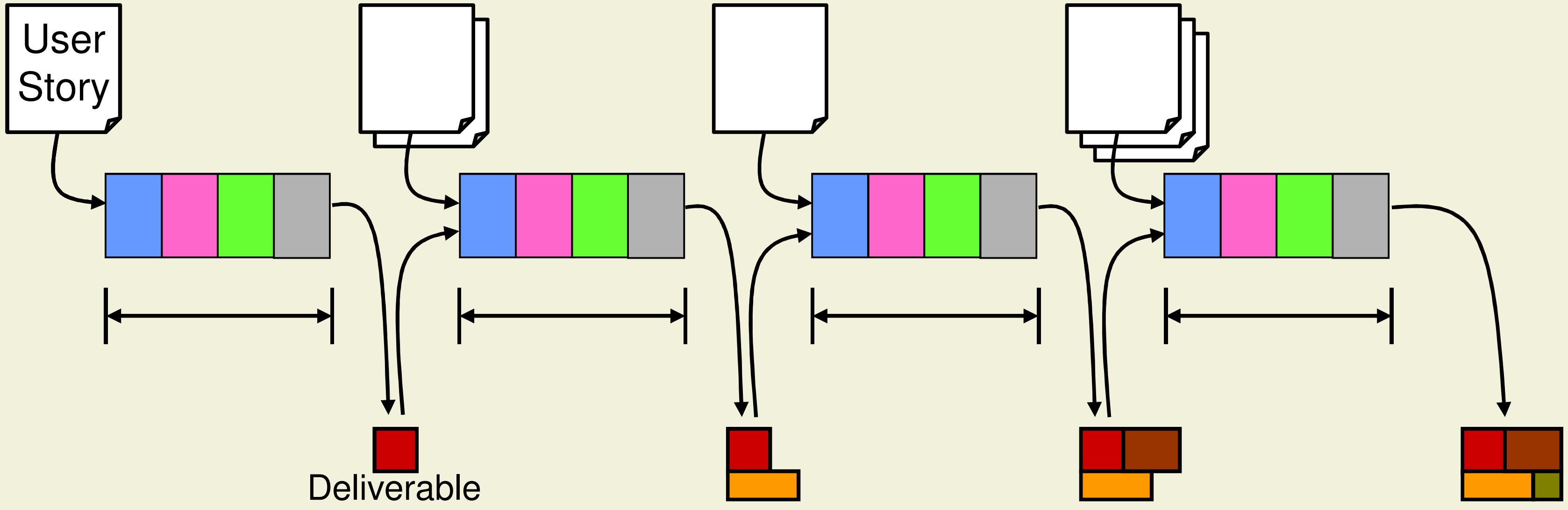
Extreme Programming: User Stories

- Brief description of functionality as viewed by a user or customer of the system
- Free form, no mandatory syntax, lightweight



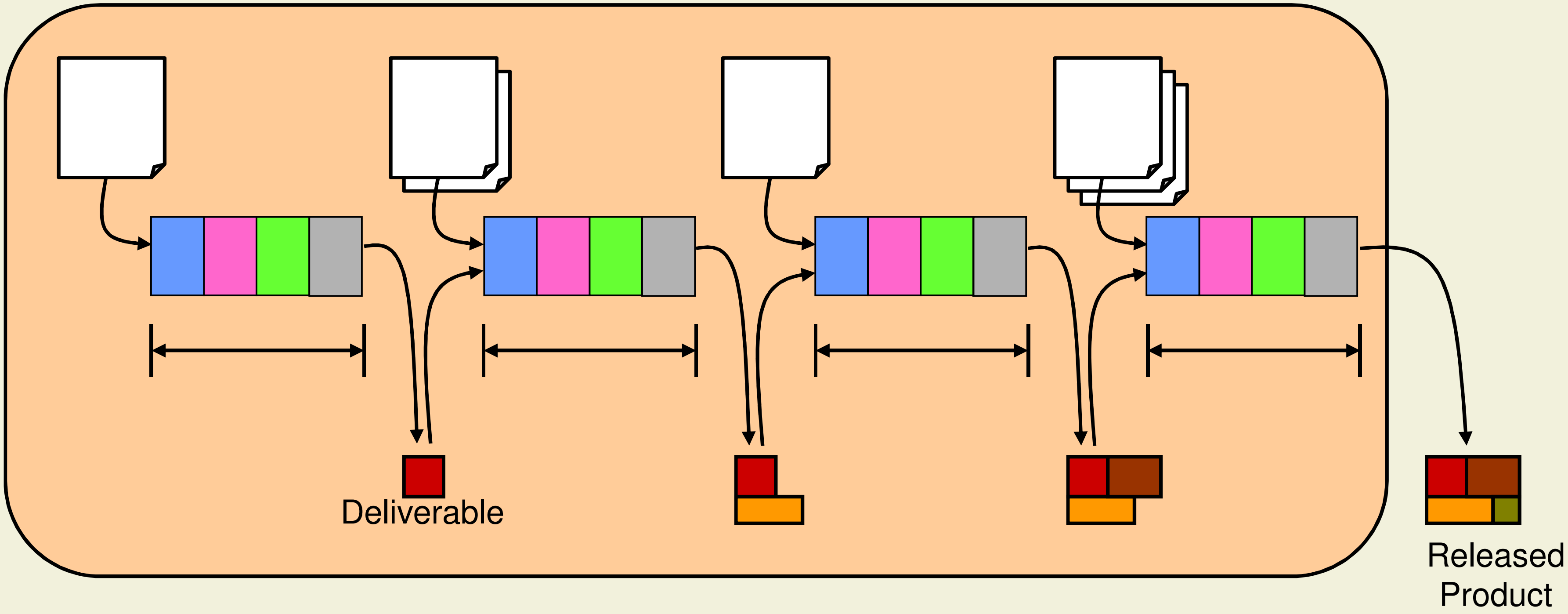
Extreme Programming: Deliverables

- Each iteration results in a deliverable
- Coded, tested, and **potentially shippable**
- Small addition of functionality



Extreme Programming: Releases

- Consists of iterations that add related functionality
- Usually every 2 to 6 months
- User can see big improvement over last release



The Basics

- XP relies on **12 principles** that are used as guides during the development process
- XP separates software development into **4 activities**, these are roles a software engineer can play
- XP advocates **12 practices** that describe how to approach the development process

The Twelve XP Principles

1. Rapid feedback
2. Assume simplicity
3. Incremental change
4. Embracing change
5. Quality work
6. Small initial investment
7. Concrete experiments
8. Open, honest communication
9. Accepted responsibility
10. Local adaptation
11. Travel light
12. Honest measurements

XP Principles (cont'd)

1. Rapid feedback
    - Generate feedback, interpret it and put experience in the system as frequently as possible
    - Business learns the benefits and shortcoming of the systems
    - Programmers lean how to best test, design, implement seconds/minutes instead of weeks/months
2. Assume simplicity
    - Do not design for reuse
    - Plan for today and trust your ability to add complexity in the future

XP Principles (cont'd)

3. Incremental change
    - Designs change a little at a time
    - Plans change a little at a time
    - Teams change a little at a time

4. Embracing change
    - Best strategies preserve most options while solving the pressing problems

XP Principles (cont’d)

5. Quality of work
    - Quality is not a free variable: the only possible values are “excellent and “insanely excellent”
6. Small initial investment
    - Tight budgets force programmers and customers to focus on essentials
    - Avoid comfort
7. Concrete experiments
    - Every abstract decision should be tested
    - The result of a design session should be a series of experiments

XP Principles (cont'd)

8. Open, honest communication
    - Deliver the bad news early
9. Accepted responsibility
    - Responsibilities should not be given, they should be accepted
10. Local adaptation
    - There are no fixed rules
11. Travel light
    - Keep things small, maintain only the essential
12. Honest measurements
    - Strive for accurate measurement of productivity

The Four XP Activities

Listening

Designing

Coding

Testing

XP Activities: Coding

- Coding as learning
- Coding as communication
- Code as end result
- Code as specification

XP Activities: Testing

- Anything that cannot be **measured** does not exist
- Without test, software is useless
- Tests are not only for **functional requirements** they are also for **performance** and adherence to **standards**
- “test infected” – **do not code before having tests**
- Write only tests that could possibly fail (but beware about that possibly)
- Tests keep the program alive longer
- Testing improves productivity

XP Activities: Listening

- Listening to customers
- Find rules that encourage useful communication
- Find rules that discourage useless communication

XP Activities: Designing

- Organize the logic of the system
- Good design ensures that every piece of logic has **only one home**
- Good design allows the extension of the system with **changes in only one place**
- Bad design is seen when one modification requires many changes
- Complexity is a source of bad design
- Design is a **daily activity of all programmers**

The Twelve XP Practices

1. The Planning Game
2. Small releases
3. Metaphor
4. Simple design
5. Testing
6. Refactoring
7. Pair programming
8. Collective ownership
9. Continuous integration
10. 40-hour week
11. On-site customer
12. Coding standards

XP Practices: 1. The Planning Game

- **Business people** decide about
  - Scope
  - Priority of features
  - Composition of releases
  - Dates of releases

- **Technical people** decide about
  - Estimates
  - Process
  - Detailed scheduling

XP Practices: 2. Small Releases

- **Working system early**
- Releases anywhere from daily to monthly
- Metaphor
  - System shape defined by a metaphor shared by the customer and programmers

XP Practices: 3. Metaphor

- A **story** that customer, programmers, and managers can tell about how the system works
- Every project is guided by a single overarching metaphor
- Vocabulary should be consistent with the metaphor
- Give a coherent story within which to work, a story that can be easily shared by business and technical
- A metaphor is a system architecture that is easy to communicate

XP Practices: 4. Simple Design

- The right design is one that:
  - **Runs all tests**
  - **Communicates everything** the programmers want to communicate
  - Contains **no duplicate code**
  - Has the **fewest possible classes** and methods
  - Say everything **once and only once**

XP Practices: 5. Testing

- Any feature without an **automated test** does not exist
- Programmers write **unit tests**
- Customers write **functional tests**
- Write test only for method that could possibly break

XP Practices: 6. Refactoring

- A change that leaves system behavior unchanged, but **enhances simplicity, flexibility**, **understandability**, and/or **performance**
- Before changing the program: Is there a way of modifying the program to **make adding** this **new feature easier**?
- After changing the program: Is there a way to **make** the **program simpler**?
- You refactor only when the systems requires you to
- **Keep all tests running**

XP Practices: 7. Pair Programming

- All production code is written with **two people** looking **at one machine**
- There are **two roles** in each pair:
  - One partner is thinking about implementation
  - The other is thinking strategically (Is this whole approach going to work? What test cases may fail? Can we simplify the system to make this problem go away?)
- Pair programming is dynamic, **different pairs** each time
- Pair programming **spreads knowledge**

XP Practices: 8. Collective Ownership

- Anybody who sees an opportunity to add value to any portion of the code is required to do so at any time
- Chaos is adverted by testing

XP Practices: 9. Continuous Integration

- Code is **integrated and tested several times a day**
- Integration ends when 100% of tests are passed

XP Practices: 10. 40 Hour Weeks

- Be fresh and rested
- Overtime is a symptom of serious problems on the project

XP Practices: 11. On site customer

- Real customers are need full time
- Provide instant feedback
- Keep development on track

XP Practices: 12. Coding standards

- The standard is indispensable
- It should not be possible to tell who wrote a piece of code
- The standard must be accepted by the whole team

XP: Discussion

- Situations where XP is not appropriate (according to Kent Beck)
    - When it is not supported by the company culture
    - More than 10 or 20 programmers (!)
    - Project too big for regular complete integration
    - Where it inherently takes a long time to get feedback
    - Where you can't realistically test (e.g., already in production using a $1,000,000 machine that is already at full capacity)