Chapter 17/16.4

Rapid Software Development / Prototyping
Objectives

- To explain how an iterative, incremental development process can lead to faster delivery of more useful software
- To discuss the essence of agile development methods
- To explain the principles and practices of extreme programming

(Cont’d)
Objectives (cont'd)

- To explain the roles of prototyping in the software process
- To explain the need for user interface prototyping (16.4)
Topics covered

- Rapid Software Development
- Incremental software development
- Agile methods
- Extreme programming
- Other rapid development approaches: RAD environments, visual programming, software reuse
- Software prototyping
Rapid Software Development
Rationale for rapid software development

- Quickly changing global markets mean businesses must be responsive to new opportunities and competition.
- Thus, rapid development and delivery is often the most critical requirement.
- Businesses may even be willing to accept lower quality if rapid delivery of essential functionality is possible.
Requirements instability

- Being responsive to changing environments = coping with unstable requirements.
- In many such cases:
  - a waterfall model of development is impractical
  - evolutionary development based on iterative specification and delivery is the only way to deliver software quickly.
Characteristics of rapid development processes

- Processes of specification, design and implementation are concurrent.
- No detailed specification and design documentation is minimal.
- System may* be developed as a series of stand-alone increments. (Users evaluate increments and make proposals for later increments.) – introduced in Chap 4.

*If this is not practical, a “throw-away prototyping” approach may be employed.
Incremental Software Development
“Incremental Software Development”

1. Define system deliverables
2. Design system architecture
3. Specify system increment
4. Build system increment
5. Validate increment
6. Deliver final system
7. System complete?
8. Validate system
9. Integrate increment
Advantages of \textit{incremental software development}\textsuperscript{*}

- \textbf{Accelerated delivery of high-priority customer services.}
  - Each increment incorporates the (next) highest priority functionality.

- \textbf{More user involvement in development.}
  - System is \textit{more likely to meet requirements} and users are more committed to system success.

\textsuperscript{*} Note: Sommerville sometimes uses the term \textit{exploratory development} for \textit{incremental software development}. 

---

\textsuperscript{*} Sommerville refers to the approach of developing software incrementally, providing benefits such as accelerated delivery of high-priority services and increased user involvement, which enhances the likelihood of meeting requirements and user commitment to the system's success. Sommerville can also use the term "exploratory development" to describe the same process, highlighting the iterative nature of software creation in this context.
Problems with iterative development and incremental delivery (= “incremental development”)

- Management problems
  - Progress can be hard to assess (lack of regular deliverables). **poor process visibility**
  - May require unfamiliar technologies or special skills.

- Contractual problems
  - Normal contractual model requires a requirements specification.

(Cont’d)
Problems with iterative development and incremental delivery (cont'd)

- **V&V problems**
  - Without a specification, what is the system being tested against?

- **Maintenance problems**
  - Continual change tends to corrupt software structure.
  - Documentation is lacking.

- **And recall:** it may be difficult to partition requirements into stand-alone increments.
  (Cont’d)
Problems with iterative development and incremental delivery (cont'd)

- For some large systems, **evolutionary / incremental development may be impractical**…
  - when **multiple teams are working at different sites**
  - when **high reliability or safety** is required
  - when **maintainability** is paramount

(Cont’d)
Problems with iterative development and incremental delivery (cont'd)

- In such cases, **throw-away prototyping**, where an experimental system is developed as **a basis for formulating the requirements** may be used.
- This system is **“thrown away”** when the requirements have been validated.
- We consider throw-away and other types of prototyping in detail later...
Sommerville’s sometimes confusing RSD-related terminology

- Two types of *Evolutionary Development*: (from Chap 4)
  1. *Throw-Away Prototyping*
  2. *Exploratory Development*, aka: (from Chap 17)
     a. *iterative software development*
     b. *iterative development and incremental delivery*,

- Recall that in Chap 4, the term *incremental delivery* (used alone) was described as an “in-between approach that combines the advantages” of the *waterfall model* and *evolutionary development*. 
Agile Methods
Agile methods for evolutionary / incremental development

- Dissatisfaction with overhead of waterfall method led to creation of agile methods. They:
  - Focus on **code** rather than the **design**;
  - Are based on **iterative** development;
  - Are intended to deliver working software quickly which can evolve quickly to meet changing requirements.

See:  www.agilealliance.org
## Principles of agile methods

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer involvement</td>
<td>The customer should be closely involved throughout the development process. Their role is to provide and prioritise new system requirements and to evaluate the iterations of the system.</td>
</tr>
<tr>
<td>Incremental delivery</td>
<td>The software is developed in increments with the customer specifying the requirements to be included in each increment.</td>
</tr>
<tr>
<td>People not process</td>
<td>The skills of the development team should be recognised and exploited. The team should be left to develop their own ways of working without prescriptive processes.</td>
</tr>
<tr>
<td>Embrace change</td>
<td>Expect the system requirements to change and design the system so that it can accommodate these changes.</td>
</tr>
<tr>
<td>Maintain simplicity</td>
<td>Focus on simplicity in both the software being developed and in the development process used. Wherever possible, actively work to eliminate complexity from the system.</td>
</tr>
</tbody>
</table>
Problems with agile methods

- Can be **difficult to keep the interest of customers** who are involved in the process.
- **Team members may be unsuited to the intense involvement** that characterizes agile methods.
- **Prioritizing changes can be difficult** where there are multiple stakeholders.

(Cont’d)
Problems with agile methods (cont'd)

- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other iterative development approaches.
Sommerville’s position

- Agile methods are probably best suited to small/medium-sized business systems or PC products.
- In particular, they are not well suited for dealing with:
  - large-scale development with multiple teams working at different sites
  - complex interactions with other systems
  - high security or safety applications
XP
Extreme programming (XP)

- Perhaps the **best-known and most widely used agile method**.
- Takes an “extreme” approach to iterative development:
  - New versions may be built **several times per day**
  - Increments are delivered to customers **every 2 weeks**
  - **All tests** must run successfully for every build

  “**iterative development on steroids**”
The XP release cycle

1. Select user stories for this release
2. Break down stories to tasks
3. Plan release
4. Develop/integrate/test software
5. Release software
6. Evaluate system
<table>
<thead>
<tr>
<th>Extreme programming practices 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Incremental planning</strong></td>
</tr>
<tr>
<td><strong>Small Releases</strong></td>
</tr>
<tr>
<td><strong>Simple Design</strong></td>
</tr>
<tr>
<td><strong>Test first development</strong></td>
</tr>
<tr>
<td><strong>Refactoring</strong></td>
</tr>
</tbody>
</table>
### Extreme programming practices 2

<table>
<thead>
<tr>
<th>Practice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pair Programming</strong></td>
<td>Developers work in pairs, checking each other's work and providing the support to always do a good job.</td>
</tr>
<tr>
<td><strong>Collective Ownership</strong></td>
<td>The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers own all the code. Anyone can change anything. <strong>“egoless programming”</strong></td>
</tr>
<tr>
<td><strong>Continuous Integration</strong></td>
<td>As soon as work on a task is complete it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.</td>
</tr>
<tr>
<td><strong>Sustainable pace</strong></td>
<td>Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity.</td>
</tr>
<tr>
<td><strong>On-site Customer</strong></td>
<td>A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.</td>
</tr>
</tbody>
</table>
XP and agile principles

- Incremental development is supported through **small, frequent system releases**.
- Customer involvement means **full-time customer engagement** with the team.
- Focus is on **people** – not process – through **pair programming, collective ownership**, and a process that **avoids long working hours**.

(Cont’d)
XP and agile principles (cont'd)

- Change supported through regular system releases.
- Maintaining simplicity (maintainability) through constant refactoring of code.
Requirements scenarios

- Requirements are expressed as scenarios or user stories written on cards.
- Development team breaks them down into implementation tasks.
- Tasks are the basis of schedule and cost estimates.
- Customer chooses stories for inclusion in the next release based on priorities and schedule estimates.
### Story card for document downloading

<table>
<thead>
<tr>
<th><strong>Downloading and printing an article</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it - this can either be through a subscription, through a company account or by credit card.</td>
</tr>
<tr>
<td>After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer.</td>
</tr>
<tr>
<td>You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful.</td>
</tr>
<tr>
<td>If the article is a print-only article, you cannot keep the PDF version so it is automatically deleted from your computer.</td>
</tr>
</tbody>
</table>
XP and change

- Conventional SE wisdom is *design for change* (via “information hiding”) to reduce maintenance costs.
- XP maintains this is *not worthwhile since changes cannot be reliably anticipated.*

(Which position is correct?)

(Cont’d)
XP and change (cont'd)

- (Instead,) XP proposes **constant code improvement** ("refactoring") to make changes easier when they have to be implemented.
Testing in XP

- **Test-first**: write tests **before coding**.
  - helps clarify requirements
- **Involve Users** in test development and validation.
- **Use automated test harnesses** to run all previous and new tests before each new release.
  - regression testing
Task cards for document downloading

Task 1: Implement principal workflow

Task 2: Implement article catalog and selection

Task 3: Implement payment collection

Payment may be made in 3 different ways. The user selects which way they wish to pay. If the user has a library subscription, then they can input the subscriber key which should be checked by the system. Alternatively, they can input an organisational account number. If this is valid, a debit of the cost of the article is posted to this account. Finally, they may input a 16 digit credit card number and expiry date. This should be checked for validity and, if valid, a debit is posted to that credit card account.
Test case description

Test 4: Test credit card validity

Input:
A string representing the credit card number and two integers representing
the month and year when the card expires

Tests:
Check that all bytes in the string are digits
Check that the month lies between 1 and 12 and the
year is greater than or equal to the current year
Using the first 4 digits of the credit card number,
check that the card issuer is valid by looking up the
card issuer table. Check credit card validity by submitting the card
number and expiry date information to the card
issuer

Output:
OK or error message indicating that the card is invalid
Pair programming in XP

- Programmers **work in pairs**, sitting together to develop code
  - but not the **same** pairs
- Helps develop **common ownership** of code and spreads knowledge across the team.
  - facilitates **“egoless programming”**
- Serves as an **informal, continuous review process**

(Cont’d)
Pair programming in XP (cont'd)

- Encourages refactoring since the whole team benefits.
- Measurements suggest development productivity is comparable to two people working independently (but with all the benefits of pair programming).
RAD Environments
“Rapid Application Development” (RAD) environments

- Other rapid development approaches have been around for years.
- **RAD environments** evolved from “fourth-generation languages” (4GL’s) and are designed to develop **data-intensive business applications**
- They rely on a high-level programming language integrated with a database.
A RAD environment

Database management system

DB programming language

Interface generator

Office systems

Report generator

Rapid application development environment
RAD environment tools

- Database programming language (e.g., SQL)
- Interface generator to create forms for data input and display
- Links to office applications such as spreadsheets or word processors
- Report generators
VISUAL PROGRAMMING
Visual programming with reuse

- **Scripting languages** such as Visual Basic support visual programming
- Applications are developed by creating an iconic user interface and associating components with the graphical icons.
- Large libraries of reusable components exist to support this.
- Components may be tailored to suit the specific application requirements.
Visual programming application screen
Problems with visual programming

- Difficult to coordinate team-based development
- No explicit system architecture (hidden)
- Complex dependencies between parts of the program can cause maintainability problems.
OTHER RSD APPROACHES:

COMPONENT ASSEMBLY
COTS
COMPOUND DOCUMENTS
Component assembly

- Systems are created quickly from a set of **reusable components** plus a mechanism to “glue” components together.
- Composition mechanism must include **control facilities** and a mechanism for **component communication**.
- Must take into account **availability** and **functionality** of existing components.
Reuse based Application-level rapid development: COTS

- Existing “off the shelf” applications can be configured and linked together.
- For example, a requirements management system could be built by using:
  - A database to store requirements;
  - A word processor to capture requirements and format reports; and
  - A spreadsheet for traceability management.
Compound documents

- Some applications/prototypes can be created by developing a **compound document**.
- This is a **document with active elements** (such as a spreadsheet) that allows user computations.
- Each active element has an associated **application** which is invoked when that element is selected.
- The **document itself is the integrator** for the different applications.
Application linking in compound documents

![Diagram of compound document linking different elements through applications]

- Text 1
- Table 1
- Text 2
- Text 3
- Sound 1
- Table 2
- Text 4
- Sound 2
- Text 5

Applications:
- Word processor
- Spreadsheet
- Audio player
PROTOTYPING
What is prototyping?

- Some traditional features:
  - An **iterative process** emphasizing
    - Rapid development
    - Evaluative use
    - Feedback
    - Modification
  - **Learning** (based on feedback)
  - **Consideration of alternatives**
  - **Concreteness** (a “real system” is developed and presented to real users)

(Cont’d)
What is prototyping? (cont’d)

- Boundary between prototyping and normal system development blurs when an evolutionary (e.g., *Extreme Programming*) development approach is used.
- Thus, our primary focus is *throw-away* prototyping.
Uses of prototypes

- Principal use is to help customers and developers better understand system requirements.
  - Experimentation stimulates anticipation of how a system could be used.
  - Attempting to use functions together to accomplish some task can easily reveal requirements problems.

(Cont’d)
Uses of prototypes (cont’d)

- Other potential uses:
  1. Evaluating proposed solutions for feasibility *(Experimental Prototyping)*
  2. “Back-to-back” system testing
  3. Training users before system delivery

- Prototyping is most often undertaken as a risk reduction activity.
Classifying prototypes

- **By purpose:**
  - **Throw-away prototyping** – to elicit and validate requirements
  - **Experimental prototyping** – to evaluate proposed solutions for feasibility, performance, etc.

- **horizontal vs. vertical** (breadth vs. depth)
- **mockups vs. breadboards** (form vs. function)
- **“Wizard of Oz” prototyping** (Turing test reversed)
Classifying prototypes

- By purpose:
  - Throw-away prototyping – to elicit and validate requirements
  - Experimental prototyping – to evaluate proposed solutions for feasibility, performance, etc.

- horizontal vs. vertical (breadth vs. depth)
- mockups vs. breadboards (form vs. function)
- “Wizard of Oz” prototyping (Turing test reversed)
Number of features

Vertical prototype

Horizontal prototype

points of comparable effort
Classifying prototypes

- By purpose:
  - Throw-away prototyping – to elicit and validate requirements
  - Experimental prototyping – to evaluate proposed solutions for feasibility, performance, etc.
- horizontal vs. vertical (breadth vs. depth)
- mockups vs. breadboards (form vs. function)
- “Wizard of Oz” prototyping (Turing test reversed)
Quin Tech “Self-service check-in and baggage drop-off design”

“The design was tested through a full-scale mock-up.”
Electronic circuit on a breadboard (REUK.co.uk)

“There is no need to solder anything, and the components can be moved around and the circuit modified thousands of times without damaging parts.”
Classifying prototypes

- By purpose:
  - Throw-away prototyping – to elicit and validate requirements
  - Experimental prototyping – to evaluate proposed solutions for feasibility, performance, etc.
- horizontal vs. vertical (breadth vs. depth)
- mockups vs. breadboards (form vs. function)
- “Wizard of Oz” prototyping (Turing test reversed)
The Wizard of Oz exposed…

“The truth is the Wizard was an illusion created by a man hidden behind a curtain.”
Simulation, prototyping, and scenarios

- What are the differences between prototyping and simulation?

(Cont’d)
Simulation, prototyping, and scenarios (cont’d)

- What is the connection between simulation models / prototypes, and *scenarios*?
  - Simulation models are *automatic* scenario generators.
  - Prototypes facilitate *manual* scenario generation.
Simulation, prototyping, and scenarios (cont’d)

- What is the connection between simulation models / prototypes, and *scenarios*?
  - Simulation models are *automatic* scenario generators.
  - Prototypes facilitate *manual* scenario generation.
What is the connection between simulation models / prototypes, and *scenarios*?

- Simulation models are *automatic* scenario generators.
- Prototypes facilitate *manual* scenario generation.
Prototyping benefits

- Misunderstandings are exposed.
- Difficult-to-use or confusing services are identified.
- Missing services are detected.
- Incomplete and/or inconsistent requirements are found by analysts as prototype is being developed.
- Can demo feasibility and usefulness.
- Basis for writing a system specification.
Prototyping process

What to include & what NOT to include.

- Establish prototype objectives
- Define prototype functionality
- Develop prototype
- Evaluate prototype

Prototyping plan
Outline definition
Executable prototype
Evaluation report
Throw-away prototyping

- Used to reduce requirements risk.
- Initial prototype is developed from outline requirements, delivered for experiment, and modified until risk is acceptably low.
Throw-away prototyping

Outline requirements → Develop prototype → Evaluate prototype → Specify system

Elicit/validate REQMTS

Reusable components

Develop software → Validate system → Delivered software system
Throw-away prototype delivery

Developers may be *pressurized* to deliver a throw-away prototype as the final system.

This is problematic...

- It may be impossible to meet non-functional requirements.
- The prototype is almost certainly undocumented.
- The system may be poorly structured and therefore difficult to maintain.
- Normal quality standards may not have been applied.
No, no, no! I won’t deliver the prototype to you!

Pressurizing the Developer

Air Tank

User Mgmt
Prototypes AS specifications?

- Some parts of the requirements (e.g., safety-critical functions) may be impossible to prototype and so don’t appear in the “specification.”

- An implementation has no legal standing as a contract.

- (Some) Non-functional requirements cannot be adequately represented in a system prototype.
Implementation techniques

- Various techniques may be used for developing prototypes:
  - Dynamic high-level languages
  - Database programming (RAD)
  - Component and application assembly
- These are not mutually exclusive – they are often used together.
- Visual programming is also an inherent part of most prototype development systems.
Dynamic high-level languages

- Include powerful data management facilities – often typeless and interpretive.
- Require large run-time support system – not normally used for large system development.
- Some offer excellent GUI development facilities.
- Some have an integrated support environment* whose facilities may be used in the prototype.
- Examples: Lisp (list structure based), Prolog (logic based), Smalltalk (object-oriented), APL (matrix processing).

* Function libraries, debuggers, symbolic evaluators, etc.
Choice of prototyping language

- What is the application domain? (e.g., NLP?, matrix manipulation?)
- What user interaction is required? (text-based? Web-based?)
- What support environment comes with the language? (e.g., tools, components)

(Cont’d)
Choice of prototyping language (cont’d)

- Different parts of the system may be programmed in different languages. (However, there may be problems with language communications.)

- A multi-paradigm language (e.g., LOOPS) can reduce this problem.
User interface prototyping

- It is impossible to pre-specify the *look and feel* of a user interface in an effective way. **Prototyping is essential.**
- UI development consumes an increasing part of overall system development costs.
User interface prototyping

- **Aim is to** allow users to gain direct experience with the interface.
- **Without this**, it is impossible to judge usability.
- **May employ a two-stage process:**
  - paper prototypes are developed initially,
  - followed by a series of increasingly sophisticated automated prototypes.
Paper prototyping

- Work through scenarios using sketches of the interface.
- Use storyboards/scenarios to present a series of interactions with the system.
- Paper prototyping is a cost-effective way of getting user reactions to an interface design proposal.
User interface evaluation

- Some **evaluation** of a user interface design should be carried out to assess its suitability.
- Thorough evaluation is very expensive and impractical for most systems.
- Ideally, **an interface should be evaluated against a usability specification**. However, it is rare for such specifications to be produced.
Usability attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learnability</td>
<td>How long does it take a new user to become productive with the system?</td>
</tr>
<tr>
<td>Speed of operation</td>
<td>How well does the system response match the user’s work practice?</td>
</tr>
<tr>
<td>Robustness</td>
<td>How tolerant is the system of user error?</td>
</tr>
<tr>
<td>Recoverability</td>
<td>How good is the system at recovering from user errors?</td>
</tr>
<tr>
<td>Adaptability</td>
<td>How closely is the system tied to a single model of work?</td>
</tr>
</tbody>
</table>
Simple evaluation techniques

- Questionnaires for user feedback.
- Video recording of system use and subsequent tape evaluation. ("protocol analysis")
- Instrumentation of code to collect information about patterns of use and user errors.
- Including code in system to collect on-line user feedback.
Key points

- An **iterative approach** to software development leads to **faster delivery of software**.

- **Agile methods** are iterative development methods that aim to **reduce development overhead and so produce software faster**.

- **Extreme programming** includes practices such as systematic testing, continuous improvement, and customer involvement.

(Cont’d)
Key points (cont’d)

- Testing in XP is a particular strength since tests are developed before code is written.
- Rapid Application Development (RAD) environments include database programming languages, form generation tools, and links to office applications.
- Throw-away prototyping is used to explore requirements and design options.

(Cont’d)
Key points (cont’d)

- Prototypes give end-users a **concrete impression** of a system’s capabilities.
- **Rapid development** of prototypes is essential. This usually requires **leaving out functionality or relaxing non-functional constraints**.

(Cont’d)
Key points (cont’d)

- Prototyping techniques include the use of very high-level languages, database programming and prototype construction from reusable components.

- Prototyping is essential for parts of the system such as the user interface which cannot be effectively pre-specified.

- Users must be involved in prototype evaluation.
Chapter 17/16.4

Rapid Software Development / Prototyping