**Key Construction Decisions**

Upstream Prerequisites,” discussed the software equivalent of blueprintsand construction permits. You might not have had much control over those preparations,so the focus of that chapter was on assessing what you have to work with whenconstruction begins. This chapter focuses on preparations that individual programmersand technical leads are responsible for, directly or indirectly.

**■ 4.1 Choice of Programming Language:**

Studies have shown that the programming-language choice affects productivity and

code quality in several ways.

1. Programmers working in a languagethey’ve used for three years or more are about 30 percent more productive thanprogrammers with equivalent experience who are new to a language

2. Programmers working with high-level languages achieve better productivity and qualitythan those working with lower-level languages. Languages such as C++, Java, Smalltalk,and Visual Basic have been credited with improving productivity, reliability, simplicity,and comprehensibility by factors of 5 to 15 over low-level languages such as assemblyand C

3. Higher-level languages are more expressive than lower-level languages.

4. Some languages are better at expressing programming concepts than others

**■ 4.2 Programming Conventions:**

In high-quality software, the implementation must beconsistent with the architecture that guides it and consistent internally. That’s thepoint of construction guidelines for variable names, class names, routine names, formattingconventions, and commenting conventions.

In a complex program, architectural guidelines give the program structural balanceand construction guidelines provide low-level harmony, articulating each class as afaithful part of a comprehensive design. Any large program requires a controllingstructure that unifies its programming-language details.

**■ 4.3 Your Location on the Technology Wave:**

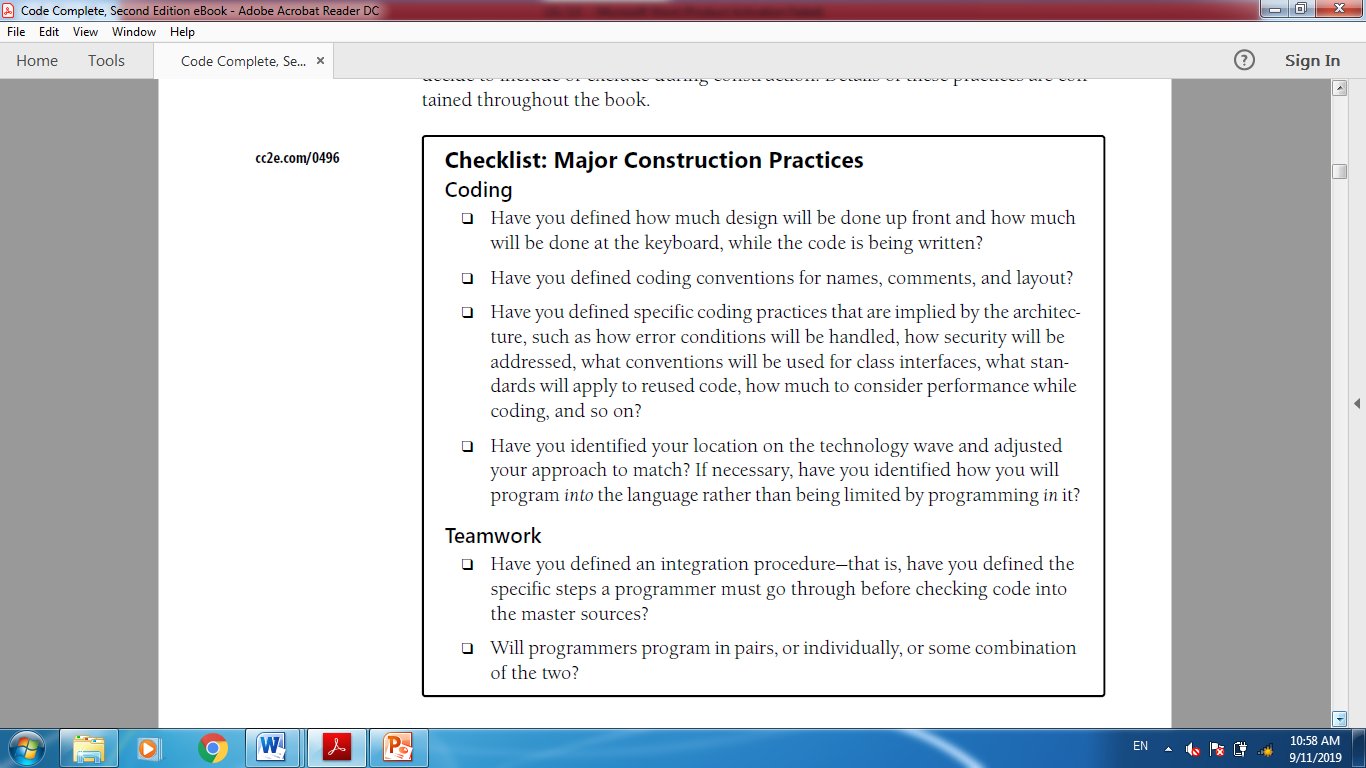
The technology cycles, or waves, imply differentprogramming practices depending on where you find yourself on the wave.

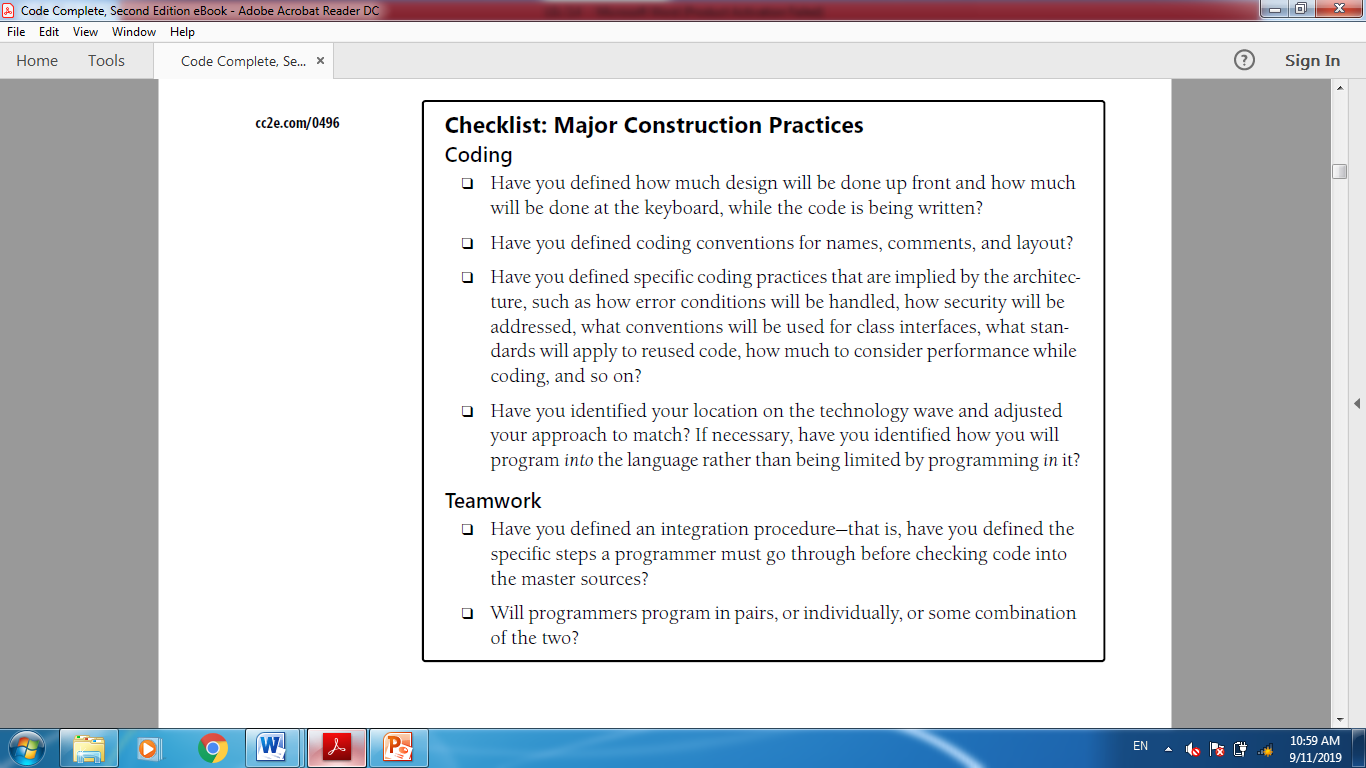
1. In early-wave environments—Few programming language choices are available, and thoselanguages tend to be buggy and poorly documented. Programmers spend significantamounts of time simply trying to figure out how the language works instead of writingnew code.. Programming tools inearly-wave environments tend to be primitiveVendors revisetheir compiler version often, and it seems that each new version breaks significantparts of your code. Tools aren’t integrated, and so you tend to work with differenttools for UI, database, reports, and business logic.

2. In mature technology environments—the end of the wave, such as Web programmingin the mid-2000s—we benefit from a rich software development infrastructure. Latewaveenvironments provide numerous programming language choices, comprehensiveerror checking for code written in those languages, powerful debugging tools,and automatic, reliable performance optimization.

**■ 4.4 Selection of Major Construction Practices:**

Part of preparing for construction is deciding which of the many available good practicesyou’ll emphasize. Some projects use pair programming and test-first development,while others use solo development and formal inspections. Either combinationof techniques can work well, depending on specific circumstances of the project.The following checklist summarizes the specific practices you should consciouslydecide to include or exclude during construction.





**Design in Construction**

**Software Design**

The process of defining the software architecture, components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements.

**Importance of Managing Complexity**

* Most technical issues are due to high complexity
* Users are demanding more functionality
* Frequent source of complexity
  + Software Crisis: Ability to produce suitable applications is not keeping pace with demand
  + Causing systems to be unreliable, slow, insecure, buggy
* “Separation of concerns” is one method to overcome complexity

**How to Overcome Complexity**

Ineffective designs come from three sources:

* + A complex solution to a simple problem
  + A Simple, incorrect solution to a complex problem
  + An inappropriate, complex solution to a complex problem

2 ways to manage complexity:

* + Minimize the amount of essential complexity
  + Keep accidental complexity from proliferating

**Desirable Design Characteristics (cont.)**

Minimal complexity

* + - KISS: Keep it simple, stupid!

Ease of maintenance

* + - Use common programming constructs and consistent naming conventions

Loose coupling

* + - Reduce interdependencies within the code

Extensibility

* + - Minimal ripple effect when changes are made

Reusability

* + - Modules, components, & subsystems can be and are reused

High fan-in

* + Highly utilized classes

Low to medium fan-out

* + Call tree not to large

Portability

* + Able to be redeployed in a different environment

Leanness

* + No extra parts

Stratification

* + Consistent levels of abstraction across subsystems

Standard techniques

* + Stay with the “tried and true”

**Levels of Design**

5 levels of abstraction in software design:

**Level 1 – Software System**

* The entire system
* Concern of the architect, not detailed designer
* Unless system is extremely small

**Level 2 – Subsystems**

* Identify all major subsystems
  + I.e. identity the architectural layers and the contents in each layer
* Major design activities at this level are
  + Partition the program into major subsystems
  + Define subsystem interfaces

**Level 3 – Classes**

* Identify all classes by subsystem
* Define class relationships
  + Generalizations
  + Dependencies
  + Associations
* For each class, define its interface

**Level 4 – Routines**

* Define the routines for each class
  + Previously defined interfaces will help
  + Need to also identify internal routines
* Level 4 activities may necessitate a return to Level 3 to further define the interface
  + This is normal and encouraged in an iterative development approach

**Level 5 – Internal Routine Design**

* Lay out the detailed functionality of each routine
* Closest activity to programming
* This includes:
  + Deciding program flow, perhaps by writing pseudocode
  + Choosing algorithms
  + Determining program calls
  + Determining return points
  + Inserting programming constructs such as loops and case statements

**Design Building Blocks: Heuristics**

**Find Real-World Objects**

It’s the OOP approach that focuses on identifying real-world and syntheticobjects.

The steps in designing with objects are

■ Identify the objects and their attributes (methods and data).

■ Determine what can be done to each object.

■ Determine what each object is allowed to do to other objects.

■ Determine the parts of each object that will be visible to other objects—whichparts will be public and which will be private.

■ Define each object’s public interface.

**Form Consistent Abstractions**

Abstraction is the ability to engage with a concept while safely ignoring some of itsdetails—handling different details at different levels.

Good programmers create abstractions at the routine-interface level, class-interfacelevel, and package-interface level—in other words, the doorknob level, door level, andhouse level—and that supports faster and safer programming.

**Encapsulate Implementation Details**

Encapsulation helps to manage complexity by forbidding youto look at the complexity

**Apply Inheritance**

In object-oriented programming,you can define a general type of employee and then define full-time employeesas general employees, except for a few differences, and part-time employees also asgeneral employees, except for a few differences. When an operation on an employeedoesn’t depend on the type of employee, the operation is handled as if the employeewere just a general employee. When the operation depends on whether the employeeis full-time or part-time, the operation is handled differently.

Defining similarities and differences among such objects is called “inheritance”because the specific part-time and

**Hide Internal Information**

Information hiding is characterized by the idea of “secrets,” design and implementation decisions that a software developer hides in one place from the rest of a program.

**Identify Areas Likely to Change**

Accommodating changes is one of the mostchallenging aspects of good program design. The goal is to isolate unstable areas sothat the effect of a change will be limited to one routine, class, or package. Here are thesteps you should follow in preparing for such perturbations.

**1. Identify items that seem likely to change.** If the requirements have been donewell, they include a list of potential changes and the likelihood of each change.

In such a case, identifying the likely changes is easy. If the requirements don’tcover potential changes, see the discussion that follows of areas that are likely tochange on any project.

**2. Separate items that are likely to change.** Compartmentalize each volatile componentidentified in step 1 into its own class or into a class with other volatilecomponents that are likely to change at the same time.

**3. Isolate items that seem likely to change.** Design the interclass interfaces to beinsensitive to the potential changes. Design the interfaces so that changes arelimited to the inside of the class and the outside remains unaffected. Any otherclass using the changed class should be unaware that the change has occurred.The class’s interface should protect its secrets.

a few areas that are likely to change:***Business rules, Hardware dependencies, Input and output, Nonstandard language features,***

**Ensure Loose Coupling**

Coupling describes how tightly a class or routine is related to other classes or routines.

The goal is to create classes and routines with small, direct, visible, and flexiblerelations to other classes and routines, which is known as “loose coupling.”

**Apply Design Patterns**

Coupling describes how tightly a class or routine is related to other classes or routines.

The goal is to create classes and routines with small, direct, visible, and flexiblerelations to other classes and routines, which is known as “loose coupling.”

* Patterns reduce complexity by providing ready-made abstractions
* Patterns reduce errors by institutionalizing details of common solutions
* Patterns provide heuristic value by suggesting design alternatives