



Chapter 1: Views and Quality Objectives of Software Construction


1.2 Quality Objectives of Software Construction

Xu Hanchuan


xhc@hit.edu.cn

February 27, 2018

Outline

- 
- **Quality properties of software systems**
 - External vs. internal quality factors
 - Important external quality factors
 - Tradeoff between quality factors
 - **Five key quality objectives of software construction**
 - **Easy to understand**: elegant and beautiful code / understandability
 - **Ready for change**: maintainability and adaptability
 - **Cheap for develop**: design for/with reuse: reusability
 - **Safe from bugs**: robustness
 - **Efficient to run**: performance
 - **Summary**

Objective of this lecture

- 
- To **know quality factors** to be cared in software construction;
 - To **understand the consequences** if quality objectives cannot be achieved;
 - To know **what construction techniques** are to be studied for each quality factor in this course.



1 Quality properties of software systems



External and internal quality factors

External quality factors affect users.

- **External quality factors:** qualities such as speed or ease of use, whose presence or absence in a software product may be detected by its users (not only the people who actually interact with the final products, but also those who purchase the software or contract out its development).
- Other qualities applicable to a software product, such as being modular, or readable, are **internal factors**, perceptible only to computer professionals who have access to the actual software text.

Internal quality factors affect the software itself and its developers.

- In the end, only external factors matter.
- But the key to achieving these external factors is in the internal ones: for the users to enjoy the visible qualities, the designers and implementers must have applied internal techniques that will ensure the hidden qualities.

External quality results from internal quality.



(1) External quality factors



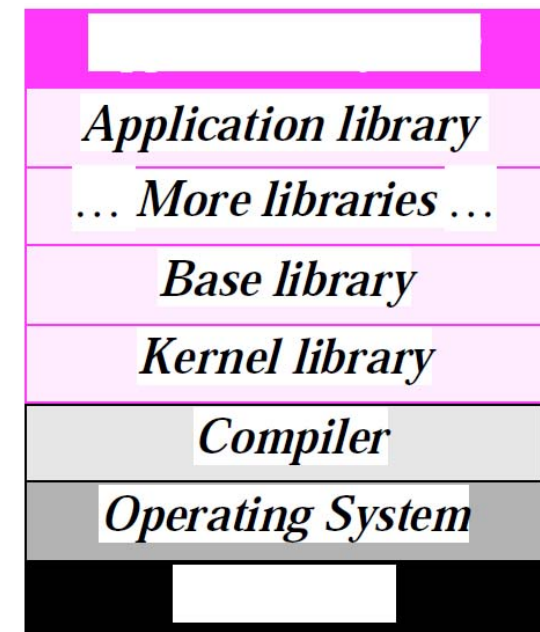
External 1: Correctness

- **Correctness is the ability of software products to perform their exact tasks, as defined by their specification.**
- **Correctness is the prime quality.**
- **Approaches of ensuring correctness: Conditional.**
 - A serious software system touches on so many areas that it would be impossible to guarantee its correctness by dealing with all components and properties on a single level. Instead, a layered approach is necessary, each layer relying on lower ones.


Assume a software system is developed in layers.

Each layer guarantees its correctness under the assumption that its lower layer is also correct.

⇒ **Reusability** (design for/ with reuse, Chapter 5)

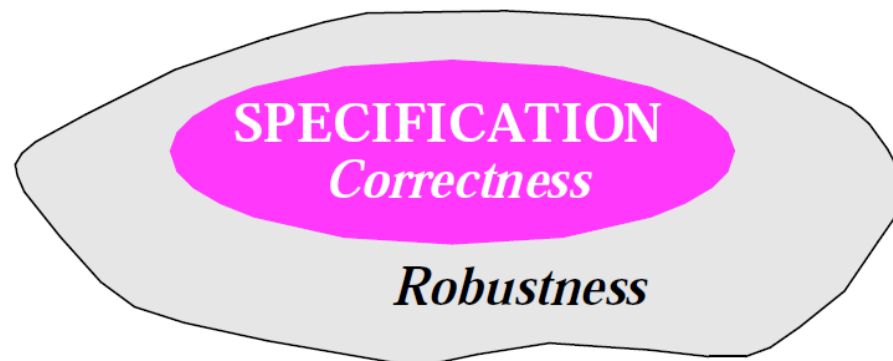


External 1: Correctness

- 
- **Approaches of ensuring correctness: Testing and debugging.**
 - **Defensive programming such as typing and assertions**, meant to help build software that is correct from the start — rather than debugging it into correctness. **⇒ Robustness (Chapter 7)**
 - **Formal approach: “check”, “guarantee” and “ensure”**
 - Mathematical techniques for formal program specification and verification**⇒ Graduate courses**

External 2: Robustness

- **Robustness is the ability of software systems to react appropriately to abnormal conditions.**
 - Robustness complements correctness.
 - Correctness addresses the behavior of a system in cases covered by its specification;
 - Robustness characterizes what happens outside of that specification.
- **Robustness is to make sure that if such cases do arise, the system does not cause catastrophic(灾难性的) events; it should produce appropriate error messages, terminate its execution cleanly, or enter a so-called “graceful degradation(毁坏)” mode.**



External 2: Robustness

- **Robustness is concerned with “abnormal case”, which implies that the notions of normal and abnormal case are always relative to a certain specification**
 - An abnormal case is simply a case that is not covered by the specification.
 - If you widen the specification, cases that used to be abnormal become normal — even if they correspond to events such as erroneous(错误的) user input that you would prefer not to happen.
 - “Normal” in this sense does not mean “desirable”, but simply “planned for in the design of the software”.
 - Although it may seem paradoxical at first that erroneous input should be called a normal case, any other approach would have to rely on subjective criteria, and so would be useless.

⇒ **Exception handling (Chapter 7)**

External 3: Extendibility

- **Extendibility** is the ease of adapting software products to changes of specification.
- **The problem of extendibility is one of scale.**
 - For small programs change is usually not a difficult issue; but as software grows bigger, it becomes harder and harder to adapt.
 - A large software system often looks to its maintainers as a giant house of cards in which pulling out any one element might cause the whole edifice to collapse.
- **We need extendibility because at the basis of all software lies some human phenomenon and hence fickleness (变化).**
- Traditional approaches did not take enough account of change, relying instead on an ideal view of the software lifecycle where an initial analysis stage freezes the requirements, the rest of the process being devoted to designing and building a solution.

External 3: Extendibility

- **Two principles are essential for improving extendibility:**
 - *Design simplicity*: a simple architecture will always be easier to adapt to changes than a complex one.
 - *Decentralization*: the more autonomous the modules, the higher the likelihood that a simple change will affect just one module, or a small number of modules, rather than triggering off a chain reaction of changes over the whole system.

⇒ Chapter 3 (ADT and OOP)

⇒ Chapter 6 (Modularity and adaptability)

External 4: Reusability

- **Reusability is the ability of software elements to serve for the construction of many different applications.**
- The need for reusability comes from the observation that software systems often follow similar patterns; it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before.
 - By capturing such a pattern, a reusable software element will be applicable to many different developments.

⇒ Chapter 5 (Design for/with reuse)

External 5: Compatibility

- **Compatibility is the ease of combining software elements with others.**
- **Compatibility is important because we do not develop software elements in a vacuum (真空): they need to interact with each other.**
- But they too often have trouble interacting because they make conflicting assumptions about the rest of the world.
 - An example is the wide variety of incompatible file formats supported by many operating systems. A program can directly use another's result as input only if the file formats are compatible.

External 5: Compatibility

- The key to compatibility lies in **homogeneity of design**, and in agreeing on standardized conventions for inter-program communication.

The key to compatibility is standardization, especially standard protocols.

- **Approaches include:**
 - **Standardized file formats**, as in the Unix system, where every text file is simply a sequence of characters.
 - **Standardized data structures**, as in Lisp systems, where all data, and programs as well, are represented by binary trees (called lists in Lisp).
 - **Standardized user interfaces**, as on various versions of Windows, OS/2 and MacOS, where all tools rely on a single paradigm for communication with the user, based on standard components such as windows, icons, menus etc.
- More general solutions are obtained by defining **standardized access protocols** to all important entities manipulated by the software.


External 6: Efficiency

- Efficiency is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.
- Efficiency does not matter much if the software is not correct (suggesting a new dictum (格言), “*do not worry how fast it is unless it is also right*”). The concern for efficiency must be balanced with other goals such as extendibility and reusability; extreme optimizations make the software so specialized as to be unfit for change and reuse.
 - Tradeoff among multiple quality factors
- Algorithms, I/O, memory management, etc.

⇒ **Chapter 8 (Performance)**

Abstract concepts for correctness of computation vs. Concrete implementation for performance through optimization

External 7: Portability (可移植性)

- 
- Portability is the ease of transferring software products to various hardware and software environments.
 - Portability addresses variations not just of the physical hardware but more generally of the **hardware-software machine**, the one that we really program, which includes the operating system, the window system if applicable, and other fundamental tools.

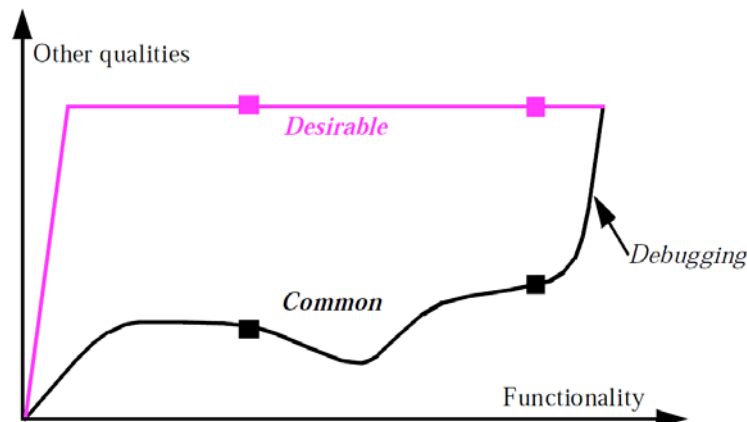
External 8: Ease of use

- Ease of use is the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. It also covers the ease of installation, operation and monitoring.
- How to provide detailed guidance and explanations to novice users, without bothering expert users who just want to get right down to business.
- One of the keys to ease of use is **structural simplicity**. A well-designed system, built according to a clear, well thought-out structure, will tend to be easier to learn and use than a messy one.
- *Know the user*. The argument is that a good designer must make an effort to understand the system's intended user community.

⇒ **Chapter 10 (GUI)**

External 9: Functionality

- **Functionality is the extent of possibilities provided by a system.**
- ***Featurism* (often “*creeping featurism*”)** 程序设计中一种不适宜的趋势，即软件开发者增加越来越多的功能，企图跟上竞争，其结果是程序极为复杂、不灵活、占用过多的磁盘空间
 - The easier problem is the **loss of consistency** that may result from the addition of new features, affecting its ease of use. Users are indeed known to complain that all the “bells and whistles” of a product’s new version make it horrendously complex.
 - The more difficult problem is to avoid being so focused on features as to forget the other qualities (**Ignorance of overall quality**).



Osmond's curves

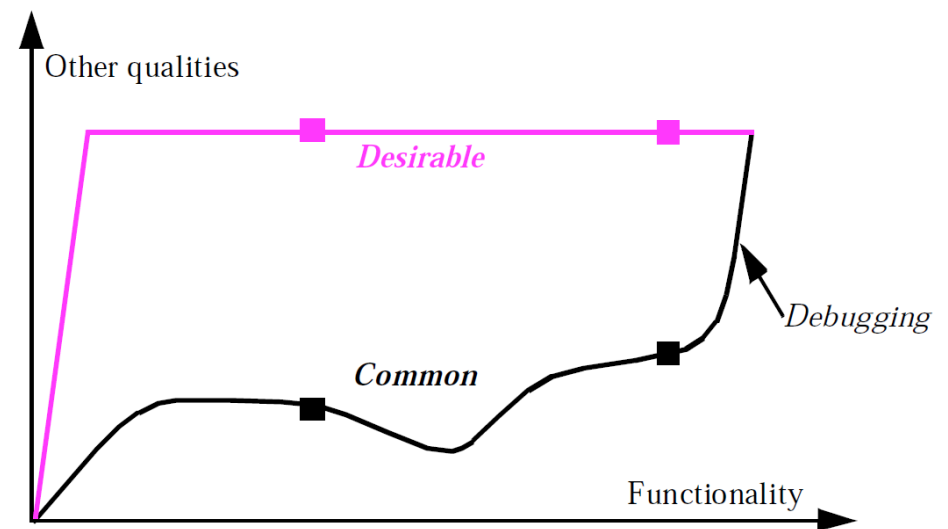
External 9: Functionality

- What Osmond suggests (the color curve) is, aided by the quality-enhancing techniques of OO development, to maintain the quality level constant throughout the project for all aspects but functionality.
- You just do not compromise on reliability, extendibility and the like: you refuse to proceed with new features until you are happy with the features you have.


⇒ **Chapter 2 (Agile, SCM)**

Start with a small set of key features with all quality factors considered.


Add more features gradually during development process and guarantee the same quality as key features.



External 10: Timeliness

- 
- **Timeliness is the ability of a software system to be released when or before its users want it.**
 - A great software product that appears too late might miss its target altogether.

External 10++: Other qualities


- 
- **Verifiability** is the ease of preparing acceptance procedures, especially test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases.
 - **Integrity** is the ability of software systems to protect their various components (programs, data) against unauthorized access and modification.
 - **Repairability** is the ability to facilitate the repair of defects.
 - **Economy**, the companion of timeliness, is the ability of a system to be completed on or below its assigned budget.



(2) Internal quality factors



Internal quality factors

- 
- Source code related factors such as Lines of Code (LOC), Cyclomatic Complexity, etc
 - Architecture-related factors such as coupling, cohesion, etc
 - Readability, understandability and clearness
 - Complexity
 - Size
 - Internal quality factors are usually used as partial measurement of external quality factors.



(3) Tradeoff between quality properties



Tradeoff between quality properties

- How can one get *integrity* without introducing protections of various kinds, which will inevitably hamper *ease of use*?
- *Economy* often seems to fight with *functionality*.
- Optimal *efficiency* would require perfect adaptation to a particular hardware and software environment, which is the opposite of *portability*, and perfect adaptation to a specification, where *reusability* pushes towards solving problems more general than the one initially given.
- *Timeliness* pressures might tempt us to use “Rapid Application Development” techniques whose results may not enjoy much *extendibility*.

Integrity vs. ease of use
Economy vs. functionality
Efficiency vs. portability
Efficiency vs. reusability
Economy vs. reusability
Timeliness vs. extendibility


Tradeoff between quality properties

- **Developers need to make tradeoffs.**
 - Too often, developers make these tradeoffs implicitly, without taking the time to examine the issues involved and the various choices available; efficiency tends to be the dominating factor in such silent decisions.
 - A true software engineering approach implies an effort to state the criteria clearly and make the choices consciously.
- **Necessary as tradeoffs between quality factors may be, one factor stands out from the rest: correctness.**
 - There is never any justification for compromising correctness for the sake of other concerns such as efficiency.
 - If the software does not perform its function, the rest is useless.

Key concerns of software construction

- All the qualities discussed above are important.
- But in the current state of the software industry, four stand out:
 - *Correctness* and *robustness*: reliability
 - Systematic approaches to software construction
 - Formal specification
 - Automatic checking during development process
 - Better language mechanism
 - Consistency checking tools
 - *Extendibility* and *reusability*: modularity

How OOP improves quality

- 
- **Correctness:** encapsulation, decentralization
 - **Robustness:** encapsulation, error handling
 - **Extendibility:** encapsulation, information hiding
 - **Reusability:** modularity, component, models, patterns
 - **Compatibility:** standardized module and interface
 - **Portability:** information hiding, abstraction
 - **Ease of use:** GUI components, framework
 - **Efficiency:** reusable components,
 - **Timeliness:** modeling, reuse
 - **Economy:** reuse
 - **Functionality:** extendibility



2 Five key quality objectives of software construction



Quality considerations of this course

- Elegant and beautiful code \Rightarrow **easy to understand**, understandability
- Design for/with reuse \Rightarrow **cheap for develop**
- Low complexity \Rightarrow **ready for changes**, easy to extend
- Robustness and correctness \Rightarrow **safe from bug**, not error-prone
- Performance and efficiency \Rightarrow **efficient to run**

Chapter 4 Understandability

Chapter 5 Reusability

Chapter 6
Extendibility/Maintainability

Chapter 7
Robustness/Correctness

Chapter 8
Efficiency/Performance

Chapter 3 ADT and OOP

Chapter 9 Refactoring

Understandability

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	代码的可理解性 (变量/子程序/ 语句的命名与构造标准、代码布局与风格、注释、复杂度) Review; Walkthrough; 函数规约	构件/项目的可理解性 (包的组织、文件的组织、命名空间)	Refactoring	
Run-time			Log Trace	

Reusability

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	ADT/OOP; 接口与实现分离; 重载; 继承/重载/重写; 组合/代理; 多态; 子类型与泛型编程; OO设计模式	API design; Library; Framework (for/with reuse)		
Run-time				

Maintainability and Adaptability

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	模块化设计; 聚合度/耦合度; SOLID; OO设计模式; Table-based programming; State-based programming; Grammar-based programming	SOLID; GRASP		SCM Version Control
Run-time				

Robustness

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	Error handling; Exception handling; Assertion; Defensive programming; Test-first programming	Unit Testing Integration Testing	Regression Testing	
Run-time	Debug Dumping		Logging Tracing	

Performance

	Moment		Period	
	Code-level	Component-level	Code-level	Component-level
Build-time	Design pattern			
Run-time	空间复杂性（内存管理）； 时间复杂性（算法性能）； 代码调优	分布式系统	Performance analysis and tuning	并行/多线程程序



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Summary



Summary

- **Quality properties of software systems**
 - External vs. internal quality factors
 - Important external quality factors
 - Tradeoff between quality factors
- **Five key quality objectives of software construction**
 - **Easy to understand**: elegant and beautiful code / understandability
 - **Ready for change**: maintainability and adaptability
 - **Cheap for develop**: design for/with reuse: reusability
 - **Safe from bugs**: robustness
 - **Efficient to run**: performance
- **Construction techniques to be studied in this course (classified by the orientation of five key quality objectives)**



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

February 27, 2018