Chapter 4
Software Measurement and Estimation

Contents

"What you measure improves."
—Donald Rumsfeld, *Known and Unknown: A Memoir*

Measurement is a process by which numbers or symbols are assigned to properties of objects. To have meaningful assignment of numbers, it must be governed by rules or theory (or, model). There are many properties of software that can be measured. Similarities can be drawn with physical objects: we can measure height, width, weight, chemical composition, etc., properties of physical objects. The numbers obtained through such measurement have little value by themselves—their key value is relative to something we want to do with those objects. For example, we may want to know the weight so we can decide what it takes to lift an object. Or, knowing physical dimensions helps us decide whether the object will fit into a certain space. Similarly, software measurement is usually done with purpose. A common purpose is for management decision making. For example, the project manager would like to be able to estimate the development cost or the time it will take to develop and deliver a software product. Similar to how knowing the object weight helps us to decide what it takes to lift it, the hope is that by measuring certain software properties we will be able to estimate the necessary development effort.

Uses of software measurements:

- Estimation of cost and effort (preferably early in the lifecycle)
- Feedback to improve the quality of design and implementation

Obviously, once a software product is already completed, we know how much effort it took to complete it. The invested effort is directly known, without the need for inferring it indirectly via some other properties of the software. However, that is too late for management decisions. Management decisions require knowing (or estimating) effort *before* we start with the development, or at least *early enough* in the process, so we can meaningfully negotiate the budget and delivery terms with the customer.

Therefore, it is important to understand correctly what measurement is about:

Measured property → [model for estimation] → Estimated property

(e.g., number of functional features) (e.g., development effort required)

Notice also that we are trying to infer properties of one entity from properties of another entity: the entity the properties of which are measured is *software* (design documents or code) and the entity the properties of which are estimated is *development process* (people's effort). The "estimation model" is usually based on empirical evidence; that is, it is derived based on observations of past projects. For past projects, both software and process characteristics are known. From this, we can try to calculate the correlation of, say, the number of functional features to, say, the development effort required. If correlation is high across a range of values, we can infer that the number of functional features is a good predictor of the development effort required. Unfortunately, we know that correlation does not equal causation. A *causal model*, which not only establishes a relationship, but also explains why, would be better, if possible to have.

Feedback to the developer is based on the knowledge of "good" ranges for software modules and systems: if the measured attributes are outside of "good" ranges, the module needs to be redesigned. It has been reported based on many observations that maintenance costs run to about 70 % of all lifetime costs of software products. Hence, good design can not only speed up the initial development, but can significantly affect the maintenance costs.

Most commonly measured characteristics of software modules and systems are related to its size and complexity. Several software characteristics were mentioned in Section 2.5, such as coupling and cohesion, and it was argued that "good designs" are characterized by "low coupling" and "high cohesion." In this chapter I will present some techniques for measuring coupling and cohesion and quantifying the quality of software design and implementation. A ubiquitous size measure is the number of lines of code (LOC). Complexity is readily observed as an important characteristic of software products, but it is difficult to operationalize complexity so that it can be measured.

taking a well-reasoned, thoughtful approach that goes beyond the simplest correlative relationships between the most superficial details of a problem.

Although it is easy to agree that more complex software is more difficult to develop and maintain, it is difficult to operationalize complexity so that it can be measured. The reader may already be familiar with *computational complexity* measure big O (or big Oh), $O(n)$. $O(n)$ measures software complexity from the machine's viewpoint in terms of how the size of the input data affects an algorithm's usage of computational resources (usually running time or memory). However, the kind of complexity measure that we need in software engineering should measure complexity form the viewpoint of human developers.

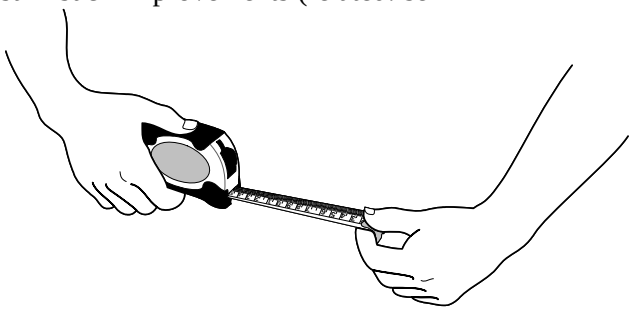
4.1 Fundamentals of Measurement Theory

"It is better to be roughly right than precisely wrong." —John Maynard Keynes

The Hawthorne effect - an increase in worker productivity produced by the psychological stimulus of being singled out and made to feel important. The Hawthorne effect describes a temporary change to behavior or performance in response to a change in the environmental conditions. This change is typically an improvement. Others have broadened this definition to mean that people's behavior and performance change following any new or increased attention.

Individual behaviors may be altered because they know they are being studied was demonstrated in a research project (1927–1932) of the Hawthorne Works plant of the Western Electric Company in Cicero, Illinois.

Initial improvement in a process of production caused by the obtrusive observation of that process. The effect was first noticed in the Hawthorne plant of Western Electric. Production increased not as a consequence of actual changes in working conditions introduced by the plant's management but because management demonstrated interest in such improvements (related: self-fulfilling hypothesis).



4.1.1 Measurement Theory

Measurement theory is a branch of applied mathematics. The specific theory we use is called the *representational theory of measurement*. It formalizes our intuitions about the way the world actually works.

Measurement theory allows us to use statistics and probability to understand quantitatively the possible variances, ranges, and types of errors in the data.

Measurement Scale

In measurement theory, we have *five types of scales*: nominal, ordinal, interval, ratio, and absolute.

In **nominal scale** we can group subjects into different categories. For example, we designate the weather condition as "sunny," "cloudy," "rainy," or "snowy." The two key requirements for the categories are: jointly exhaustive and mutually exclusive. Mutually exclusive means a measured attribute can be classified into one and only one category. Jointly exhaustive means that all categories together should cover all possible values of the attribute. If the measured attribute has more categories than we are interested in, an "other" category can be introduced to make the categories jointly exhaustive. Provided that categories are jointly exhaustive and mutually exclusive, we have the minimal conditions necessary for the application of statistical analysis. For example, we may want to compare the values of software attributes such as defect rate, cycle time, and requirements defects across the different categories of software products.

Ordinal scale refers to the measurement operations through which the subjects can be compared in order. An example ordinal scale is: "bad," "good," and "excellent," or "star" ratings used for products or services on the Web. An ordinal scale is asymmetric in the sense that if $A > B$ is true then $B > A$ is false. It has the transitivity property in that if $A > B$ and $B > C$, then $A > C$. Although ordinal scale orders subjects by the magnitude of the measured property, it offers no information

about the relative magnitude of the difference between subjects. For example, we only know that "excellent" is better than "good," and "good" is better than "bad." However, we cannot compare that the relative differences between the excellent-good and good-bad pairs. A commonly used ordinal scale is an *n*-point Likert scale, such as the Likert five-point, seven-point, or ten-point scales. For example, a five-point Likert scale for rating books or movies may assign the following values: 1 = "Hated It," 2 = "Didn't Like It," 3 = "Neutral," 4 = "Liked It," and 5 = "Loved It." We know only that 5 > 4, 4 > 3, 5 > 2, etc., but we cannot say how much greater is 5 than 4. Nor can we say that the difference between categories 5 and 4 is equal to that between 3 and 2. This implies that we cannot use arithmetic operations such as addition, subtraction, multiplication and division. Nonetheless, the assumption of equal distance is often made and the average rating reported (e.g., product rating at Amazon.com uses fractional values, such as 3.7 stars).

Interval scale indicates the exact differences between measurement points. An interval scale requires a well-defined, fixed unit of measurement that can be agreed on as a common standard and that is repeatable. A good example is a traditional temperature scale (centigrade or Fahrenheit scales). Although the zero point is defined in both scales, it is arbitrary and has no meaning. Thus we can say that the difference between the average temperature in Florida, say 80°F, and the average temperature in Alaska, say 20°F, is 60°F, but we do not say that 80°F is four times as hot as 20°F. The arithmetic operations of addition and subtraction can be applied to interval scale data.

Ratio scale is an interval scale for which an absolute or nonarbitrary zero point can be located. Absolute or true zero means that the zero point represents the absence of the property being measured (e.g., no money, no behavior, none correct). Examples are mass, temperature in degrees Kelvin, length, and time interval. Ratio scale is the highest level of measurement and all arithmetic operations can be applied to it, including division and multiplication.

For interval and ratio scales, the measurement can be expressed in both integer and noninteger data. Integer data are usually given in terms of frequency counts (e.g., the number of defects that could be encountered during the testing phase).

Absolute scale is used when there is only one way to measure a property. It is independent of the physical properties of any specific substance. In practice, values on an absolute scale are usually (if not always) obtained by counting. An example is counting entities, such as chairs in a room.

Some Basic Measures

Ratio

Proportion

Percentage

Rate

Six Sigma

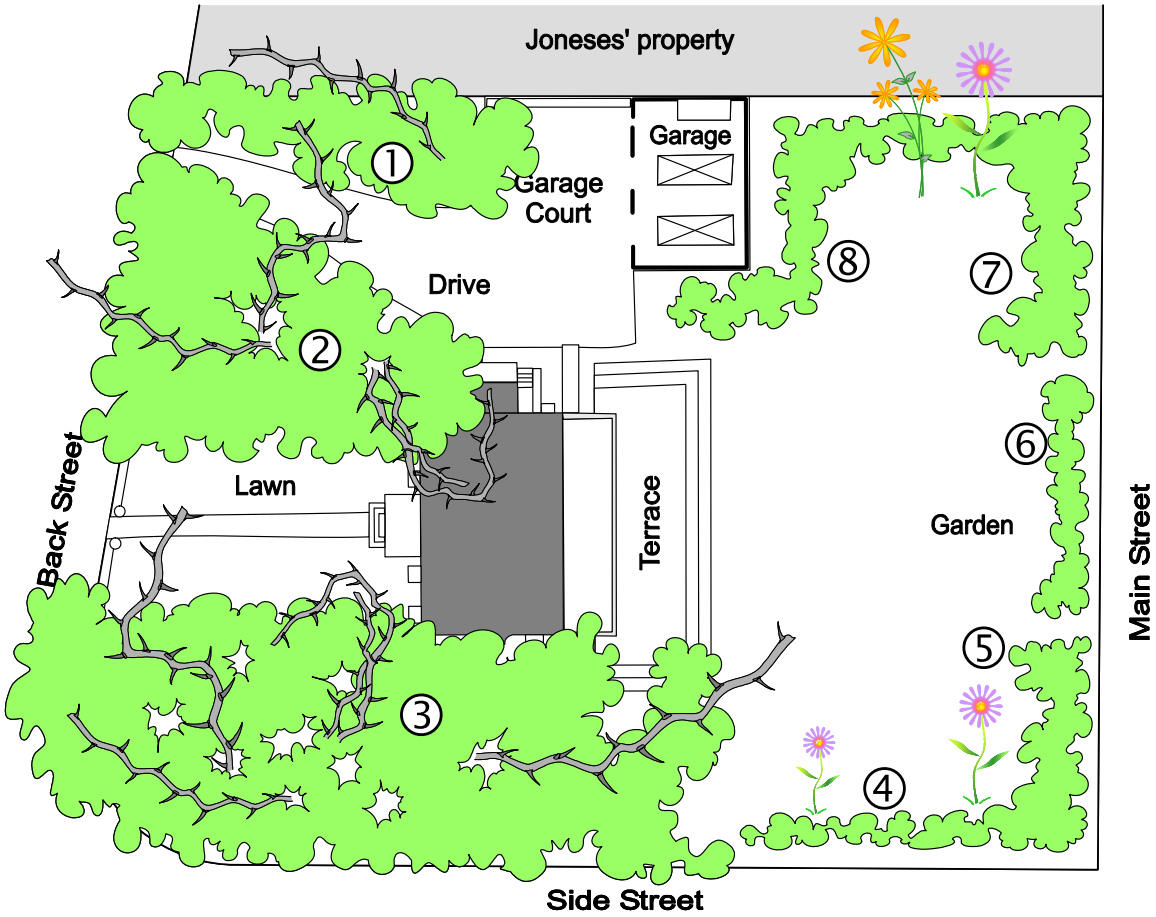


Figure 4-1: Issues with subjective size measures (compare to Figure 1-10). Left side of the hedge as seen by a pessimist; right side seen by an optimist.

4.2 What to Measure?

Given a software artifact (design document or source code), generally we can measure

1. Attributes of any *representation* or *description* of a problem or a solution. Two main categories of representations are structure vs. behavior.
2. Attributes of the *development process* or *methodology*.

Measured aspect:

- quantity (size)
- complexity

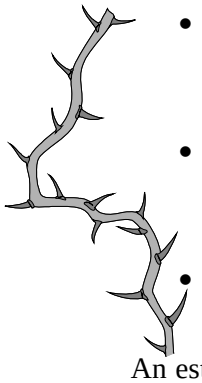
If the purpose of software measurement is estimation of cost and effort, we would like to measure at an early stage in the software life-cycle. Typically a budget allocation is set at an early phase of a procurement process and a decision on contract price made on these budget constraints and

suppliers' tender responses. Consequently, the functional decomposition of the planned system needs to be at a high level, but must be of sufficient detail to flush out as many of the implied requirements and hidden complexities as possible, and as early as possible. In the ideal world this would be a full and detailed decomposition of the use cases, but this is impractical during the estimation process, because estimates need to be produced within tight time frames.

Figure 4-1

4.2.1 Use Case Points

Intuitively, projects with many complicated requirements take more effort to design and implement than projects with few simple requirements. In addition, the effort depends not only on inherent difficulty or complexity of the problem, but also on what tools the developers employ and how skilled the developers are. The factors that determine the time to complete a project include:



- *Functional requirements*: These are often represented with use cases (Section 2.3). The complexity of use cases, in turn, depends on the number and complexity of the actors and the number of steps (transactions) to execute each use case.
- *Nonfunctional requirements*: These describe the system's nonfunctional properties, known as FURPS+ (see Section 2.2.1), such as security, usability, and performance. These are also known as the "technical complexity factors."
- *Environmental factors*: Various factors such as the experience and knowledge of the development team, and how sophisticated tools they will be using for the development.

An estimation method that took into account the above factors early in a project's life cycle, and produced a reasonable accurate estimate, say within 20% of the actual completion time, would be very helpful for project scheduling, cost, and resource allocation.

Because use cases are developed at the earliest or notional stages of system design, they afford opportunities to understand the scope of a project early in the software life-cycle. The *Use Case Points (UCP)* method provides the ability to estimate the person-hours a software project requires based on its use cases. The UCP method analyzes the use case actors, scenarios, nonfunctional requirements, and environmental factors and abstracts them into an equation. Detailed use case descriptions (Section 2.3.3) must be derived before the UCP method can be applied. The UCP method cannot be applied to sketchy use cases. As discussed in Section 2.3.1, we can apply user story points (described in Section 2.2.3) for project effort estimation at this very early stage.

The formula for calculating UCP is composed of three variables:

1. *Unadjusted Use Case Points (UUCP)*, which measures the complexity of the functional requirements
2. The *Technical Complexity Factor (TCF)*, which measures the complexity of the nonfunctional requirements
3. The *Environment Complexity Factor (ECF)*, which assesses the development team's experience and their development environment

Table 4-1: Actor classification and associated weights.

Actor type	Description of how to recognize the actor type	Weight
Simple	The actor is another system which interacts with our system through a defined application programming interface (API).	1
Average	The actor is a person interacting through a text-based user interface, or another system interacting through a protocol, such as a network communication protocol.	2
Complex	The actor is a person interacting via a graphical user interface.	3

Each variable is defined and computed separately using weighted values, subjective values, and constraining constants. The subjective values are determined by the development team based on their perception of the project's technical complexity and the team's efficiency. Here is the equation:

$$UCP = UUCP \times TCF \times ECF \quad (4.1)$$

Unadjusted Use Case Points (UUCPs) are computed as a sum of these two components:

1. The *Unadjusted Actor Weight (UAW)*, based on the combined complexity of all the actors in all the use cases.
2. The *Unadjusted Use Case Weight (UUCW)*, based on the total number of activities (or steps) contained in all the use case scenarios.

The computation of these components is described next.

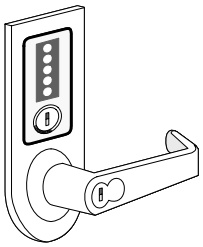
Unadjusted Actor Weight (UAW)

An actor in a use case might be a person, another program, a piece of hardware, etc. The weight for each actor depends on how sophisticated is the interface between the actor and the system. Some actors, such as a user working with a text-based command-line interface, have very simple needs and increase the complexity of a use case only slightly. Other actors, such as a user working with a highly interactive graphical user interface, have a much more significant impact on the effort to develop a use case. To capture these differences, each actor in the system is classified as simple, average, or complex, and is assigned a weight as shown in Table 4-1. This scale for rating actor complexity was devised by expert developers based on their experience. Notice that this is an *ordinal scale* (Section 4.1.1). You can think of this as a scale for "star rating," similar to "star ratings" of books (Amazon.com), films (IMDb.com), or restaurants (yelp.com). Your task is, using this scale, to assign "star ratings" to all actors in your system. In our case, we can assign one, two, or three "stars" to actors, corresponding to "Simple," "Average," or "Complex" actors, respectively. Table 4-2 shows my ratings for the actors in the case study of home access control, for which the actors are described in Section 2.3.1. The UAW is calculated by totaling the number of actors in each category, multiplying each total by its specified weighting factor, and then adding the products we obtain:

Table 4-2: Actor classification for the case study of home access control (see Section 2.3).

Actor name	Description of relevant characteristics	Complexity	Weight
Landlord	Landlord is interacting with the system via a graphical user interface (when managing users on the central computer).	Complex	3
Tenant	Tenant is interacting through a text-based user interface (assuming that identification is through a keypad; for biometrics based identification methods Tenant would be a complex actor).	Average	2
LockDevice	LockDevice is another system which interacts with our system through a defined API.	Simple	1
LightSwitch	Same as LockDevice.	Simple	1
AlarmBell	Same as LockDevice.	Simple	1
Database	Database is another system interacting through a protocol.	Average	2
Timer	Same as LockDevice.	Simple	1
Police	Our system just sends a text notification to Police.	Simple	1

$$UAW(\text{home access}) = 5 \times \text{Simple} + 2 \times \text{Average} + 1 \times \text{Complex} = 5{\times}1 + 2{\times}2 + 1{\times}3 = 12$$



Unadjusted Use Case Weight (UUCW)

The UUCW is derived from the number of use cases in three categories: simple, average, and complex (see Table 4-3). Each use case is categorized based on the number of steps (or, transactions) within its event flow, including both the main success scenario and alternative scenarios (extensions).

The number of steps in a scenario affects the estimate. A large number of steps in a use case scenario will bias the UUCW toward complexity and increase the UCPs. A small number of steps will bias the UUCW toward simplicity and decrease the UCPs. Sometimes, a large number of steps can be reduced without affecting the business process.

The UUCW is calculated by tallying the number of use cases in each category, multiplying each total by its specified weighting factor, and then adding the products. For example, Table 4-4 computes the UUCW for the sample case study.

There is a controversy on how to count alternate scenarios (extensions). Initially, it was suggested to ignore all scenarios except the main success scenario. The current view is that extensions represent a significant amount of work and need to be included in effort estimation. However, it is not agreed upon how to do the inclusion. The problem is that you cannot simply count the number of lines in an extension scenario and add those to the lines in the main success scenario.

Table 4-3: Use case weights based on the number of transactions.

Use case category	Description of how to recognize the use-case category	Weight
Simple	Simple user interface. Up to one participating actor (plus initiating actor). Number of steps for the success scenario: ≤ 3. If presently available, its domain model includes ≤ 3 concepts.	5
Average	Moderate interface design. Two or more participating actors. Number of steps for the success scenario: 4 to 7. If presently available, its domain model includes between 5 and 10 concepts.	10
Complex	Complex user interface or processing. Three or more participating actors. Number of steps for the success scenario: ≥ 7. If available, its domain model includes ≥ 10 concepts.	15

As seen in UC-7: AuthenticateUser (Section 2.3), each extension starts with a result of a transaction, rather than a new transaction itself. For example, extension 2a ("Tenant/Landlord enters an invalid identification key") is the result of the transaction described by step 2 of the main success scenario ("Tenant/Landlord supplies an identification key"). So, item 2a in the extensions section of UC-7: AuthenticateUser is not counted. The same, of course, is true for 2b, 2c, and 3a. The transaction count for the use case in UC-7: AuthenticateUser is then ten. You may want to count 2b1 and 2b2 only once but that is more effort than is worthwhile, and they may be separate transactions sharing common text in the use case.

Another mechanism for measuring use case complexity is counting the concepts obtained by domain modeling (Section 2.4). Of course, this assumes that the domain model is already derived at the time the estimate is being made. The concepts can be used in place of transactions once it has been determined which concepts model a specific use case. As indicated in Table 4-3, a simple use case is implemented by 5 or fewer concepts, an average use case by 5 to 10 concepts, and a complex use case by more than 10 concepts. The weights are as before. Each type of use case is then multiplied by the weighting factor, and the products are added up to get the UUCW.

The UUCW is calculated by tallying the use cases in each category, multiplying each count by its specified weighting factor (Table 4-3), and then adding the products:

$$UUCW(\text{home access}) = 1 \times \text{Simple} + 5 \times \text{Average} + 2 \times \text{Complex} = 1{\times}5 + 5{\times}10 + 2{\times}15 = 85$$

The UUCP is computed by adding the UAW and the UUCW. Based on the scores in Table 4-2 and Table 4-4, the UUCP for our case study project is $UUCP = UAW + UUCW = 12 + 85 = 97$.

The UUCP gives the unadjusted size of the overall system, unadjusted because it does not account for the nonfunctional requirements (TCFs) and the environmental factors (ECFs).

Table 4-4: Use case classification for the case study of home access control (see Section 2.3).

Use case	Description	Category	Weight
Unlock (UC-1)	Simple user interface. 5 steps for the main success scenario. 3 participating actors (LockDevice, LightSwitch, and Timer).	Average	10
Lock (UC-2)	Simple user interface. 2+3=5 steps for the all scenarios. 3 participating actors (LockDevice, LightSwitch, and Timer).	Average	10
ManageUsers (UC-3)	Complex user interface. More than 7 steps for the main success scenario (when counting UC-6 or UC-7). Two participating actors (Tenant, Database).	Complex	15
ViewAccessHistory (UC-4)	Complex user interface. 8 steps for the main success scenario. 2 participating actors (Database, Landlord).	Complex	15
AuthenticateUser (UC-5)	Simple user interface. 3+1=4 steps for all scenarios. 2 participating actors (AlarmBell, Police).	Average	10
AddUser (UC-6)	Complex user interface. 6 steps for the main success scenario (not counting UC-3). Two participating actors (Tenant, Database).	Average	10
RemoveUser (UC-7)	Complex user interface. 4 steps for the main success scenario (not counting UC-3). One participating actor (Database).	Average	10
Login (UC-8)	Simple user interface. 2 steps for the main success scenario. No participating actors.	Simple	5

Technical Complexity Factor (TCF)—Nonfunctional Requirements

Thirteen standard technical factors were identified (by expert developers) to estimate the impact on productivity of the nonfunctional requirements for the project (see Table 4-5). Each factor is weighted according to its relative impact.

The development team should assess the perceived complexity of each technical factor from Table 4-5 in the context of their project. Based on their assessment, they assign another "star rating," a *perceived complexity* value between zero and five. The perceived complexity value reflects the team's subjective perception of how much effort will be needed to satisfy a given nonfunctional requirement. For example, if they are developing a distributed system (factor T1 in Table 4-5), it will require more skill and time than if developing a system that will run on a single computer. A perceived complexity value of 0 means that a technical factor is irrelevant for this project, 3 corresponds to average effort, and 5 corresponds to major effort. When in doubt, use 3.

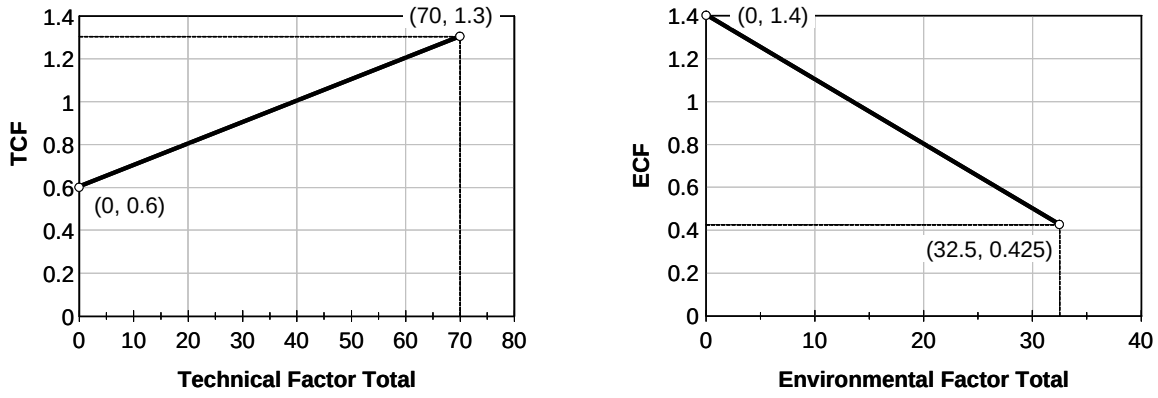
Each factor's weight (Table 4-5) is multiplied by its perceived complexity factor to produce the calculated factor. The calculated factors are summed to produce the Technical Total Factor. Table 4-6 calculates the technical complexity for the case study.

Two constants are used with the Technical Total Factor to produce the TCF. The constants limit the impact the TCF has on the UCP equation (4.1) from a range of 0.6 (when perceived complexities are all zero) to a maximum of 1.3 (when perceived complexities are all five), see Figure 4-2(a).

Table 4-5: Technical complexity factors and their weights.

Technical factor	Description	Weight
T1	Distributed system (running on multiple machines)	2
T2	Performance objectives (are response time and throughput performance critical?)	1(*)
T3	End-user efficiency	1
T4	Complex internal processing	1
T5	Reusable design or code	1
T6	Easy to install (are automated conversion and installation included in the system?)	0.5
T7	Easy to use (including operations such as backup, startup, and recovery)	0.5
T8	Portable	2
T9	Easy to change (to add new features or modify existing ones)	1
T10	Concurrent use (by multiple users)	1
T11	Special security features	1
T12	Provides direct access for third parties (the system will be used from multiple sites in different organizations)	1
T13	Special user training facilities are required	1

(*) Some sources assign 2 as the weight for the performance objectives factor (T2).



(a) (b)

Figure 4-2: Scaling constants for technical and environmental factors.

TCF values less than one reduce the UCP because any positive value multiplied by a positive fraction decreases in magnitude: 100 × 0.6 = 60 (a reduction of 40%). TCF values greater than one increase the UCP because any positive value multiplied by a positive mixed number increases in magnitude: 100 × 1.3 = 130 (an increase of 30%). The constants were determined by interviews with experienced developers, based on their subjective estimates.

Because the constants limit the TCF from a range of 0.6 to 1.3, the TCF can impact the UCP equation from −40% (0.6) to a maximum of +30% (1.3). The formula to compute the TCF is:

$$TCF = \text{Constant-1} + \text{Constant-2} \times \text{Technical Factor Total} = C_1 + C_2 \cdot \sum_{i=1}^{13} W_i \cdot F_i \qquad (4.2)$$

where,

Table 4-6: Technical complexity factors for the case study of home access (see Section 2.3).

Technical factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight×Perceived Complexity)
T1	Distributed, Web-based system, because of ViewAccessHistory (UC-4)	2	3	2×3 = 6
T2	Users expect good performance but nothing exceptional	1	3	1×3 = 3
T3	End-user expects efficiency but there are no exceptional demands	1	3	1×3 = 3
T4	Internal processing is relatively simple	1	1	1×1 = 1
T5	No requirement for reusability	1	0	1×0 = 0
T6	Ease of install is moderately important (will probably be installed by technician)	0.5	3	0.5×3 = 1.5
T7	Ease of use is very important	0.5	5	0.5×5 = 2.5
T8	No portability concerns beyond a desire to keep database vendor options open	2	2	2×2 = 4
T9	Easy to change minimally required	1	1	1×1 = 1
T10	Concurrent use is required (Section 5.3)	1	4	1×4 = 4
T11	Security is a significant concern	1	5	1×5 = 5
T12	No direct access for third parties	1	0	1×0 = 0
T13	No unique training needs	1	0	1×0 = 0
			Technical Factor Total:	31

Constant-1 (C_1) = 0.6

Constant-2 (C_2) = 0.01

W_i = weight of i^{th} technical factor (Table 4-5)

F_i = perceived complexity of i^{th} technical factor (Table 4-6)

Formula (4.2) is illustrated in Figure 4-2(a). Given the data in Table 4-6, the TCF = 0.6 + (0.01 × 31) = 0.91. According to equation (4.1), this results in a reduction of the UCP by 9%.

Environment Complexity Factor (ECF)

The environmental factors (Table 4-7) measure the experience level of the people on the project and the stability of the project. Greater experience will in effect reduce the UCP count, while lower experience will in effect increase the UCP count. One might wish to consider other external factors, such as the available budget, company's market position, the state of the economy, etc.

The development team determines each factor's perceived impact based on their perception the factor has on the project's success. A value of 1 means the factor has a strong, negative impact for the project; 3 is average; and 5 means it has a strong, positive impact. A value of zero has no impact on the project's success. For factors E1-E4, 0 means no experience in the subject, 3 means average, and 5 means expert. For E5, 0 means no motivation for the project, 3 means average, and 5 means high motivation. For E6, 0 means unchanging requirements, 3 means average amount of change expected, and 5 means extremely unstable requirements. For E7, 0 means no part-time technical staff, 3 means on average half of the team is part-time, and 5 means all of the team is

Table 4-7: Environmental complexity factors and their weights.

Environmental factor	Description	Weight
E1	Familiar with the development process (e.g., UML-based)	1.5
E2	Application problem experience	0.5
E3	Paradigm experience (e.g., object-oriented approach)	1
E4	Lead analyst capability	0.5
E5	Motivation	1
E6	Stable requirements	2
E7	Part-time staff	–1
E8	Difficult programming language	–1

part-time. For E8, 0 means an easy-to-use programming language will be used, 3 means the language is of average difficulty, and 5 means a very difficult language is planned for the project.

Each factor's weight is multiplied by its perceived impact to produce its calculated factor. The calculated factors are summed to produce the Environmental Factor Total. Larger values for the Environment Factor Total will have a greater impact on the UCP equation. Table 4-8 calculates the environmental factors for the case study project (home access control), assuming that the project will be developed by a team of upper-division undergraduate students.

To produce the final ECF, two constants are computed with the Environmental Factor Total. Similar to the TCF constants above, these constants were determined based on interviews with expert developers. The constants constrain the impact the ECF has on the UCP equation from 0.425 (part-time workers and difficult programming language = 0, all other values = 5) to 1.4 (perceived impact is all 0). Therefore, the ECF can reduce the UCP by 57.5% and increase the UCP by 40%, see Figure 4-2(b). The ECF has a greater potential impact on the UCP count than the TCF. The formula is:

$$ECF = \text{Constant-1} + \text{Constant-2} \times \text{Environmental Factor Total} = C_1 + C_2 \cdot \sum_{i=1}^{8} W_i \cdot F_i \qquad (4.3)$$

where,

Constant-1 (C_1) = 1.4

Constant-2 (C_2) = –0.03

W_i = weight of i^{th} environmental factor (Table 4-7)

F_i = perceived impact of i^{th} environmental factor (Table 4-8)

Formula (4.3) is illustrated in Figure 4-2(b). Given the data in Table 4-8, the ECF = 1.4 + (–0.03×11) = 1.07. For the sample case study, the team's modest software development experience resulted in an average EFT. All four factors E1-E4 scored relatively low. According to equation (4.1), this results in an increase of the UCP by 7%.

Table 4-8: Environmental complexity factors for the case study of home access (Section 2.3).

Environmental factor	Description	Weight	Perceived Impact	Calculated Factor (Weight× Perceived Impact)
E1	Beginner familiarity with the UML-based development	1.5	1	1.5×1 = 1.5
E2	Some familiarity with application problem	0.5	2	0.5×2 = 1
E3	Some knowledge of object-oriented approach	1	2	1×2 = 2
E4	Beginner lead analyst	0.5	1	0.5×1 = 0.5
E5	Highly motivated, but some team members occasionally slacking	1	4	1×4 = 4
E6	Stable requirements expected	2	5	2×5 = 5
E7	No part-time staff will be involved	–1	0	–1×0 = 0
E8	Programming language of average difficulty will be used	–1	3	–1×3 = –3
			Environmental Factor Total:	11

Calculating the Use Case Points (UCP)

As a reminder, the UCP equation (4.1) is copied here:

$$UCP = UUCP \times TCF \times ECF$$

From the above calculations, the UCP variables have the following values:

UUCP = 97

TCF = 0.91

ECF = 1.07

For the sample case study, the final UCP is the following:

UCP = 97 × 0.91 × 1.07 = 94.45 or 94 use case points.

Note for the sample case study, the combined effect of TCF and ECF was to increase the UUCP by approximately 3 percent (94/97×100 – 100 = +3%). This is a minor adjustment and can be ignored given that many other inputs into the calculation are subjective estimates.

Discussion of the UCP Metric

Notice that the UCP equation (4.1) is not consistent with measurement theory, because the counts are on a ratio scale and the scores for the adjustment factors are on an ordinal scale (see Section 4.1.1). However, such formulas are often used in practice.

It is worth noticing that UUCW (Unadjusted Use Case Weight) is calculated simply by adding up the perceived weights of individual use cases (Table 4-3). This assumes that all use cases are completely independent, which usually is not the case. The merit of linear summation of size measures was already discussed in Sections 1.2.5 and 2.2.3.

UCP appears to be based on a great deal of subjective and seemingly arbitrary parameters, particularly the weighting coefficients. For all its imperfections, UCP has become widely adopted because it provides valuable estimate early on in the project, when many critical decisions need to be made. See the bibliographical notes (Section 4.7) for literature on empirical evidence about the accuracy of UCP-based estimation.

UCP measures how "big" the software system will be in terms of functionality. The software size is the same regardless of who is building the system or the conditions under which the system is being built. For example, a project with a UCP of 100 may take longer than one with a UCP of 90, but we do not know by how much. From the discussion in Section 1.2.5, we know that to calculate the time to complete the project using equation (1.2) we need to know the team's *velocity*. How to factor in the team velocity (productivity) and compute the estimated number of hours will be described later in Section 4.6.

4.2.2 Cyclomatic Complexity

One of the most common areas of complexity in a program lies in complex conditional logic (or, control flow). Thomas McCabe [1974] devised a measure of *cyclomatic complexity*, intended to capture the complexity of a program's conditional logic. A program with no branches is the least complex; a program with a loop is more complex; and a program with two crossed loops is more complex still. Cyclomatic complexity corresponds roughly to an intuitive idea of the number of different paths through the program—the greater the number of different paths through a program, the higher the complexity.

McCabe's metric is based on graph theory, in which you calculate the cyclomatic number of a graph G, denoted by $V(G)$, by counting the number of linearly independent paths within a program. Cyclomatic complexity is

$$V(G) = e - n + 2 \tag{4.4}$$

where e is the number of edges, n is the number of nodes.

Converting a program into a graph is illustrated in Figure 4-3. It follows that cyclomatic complexity is also equal to the number of binary decisions in a program plus 1. If all decisions are not binary, a three-way decision is counted as two binary decisions and an n-way case (select or switch) statement is counted as $n - 1$ binary decisions. The iteration test in a looping statement is counted as one binary decision.

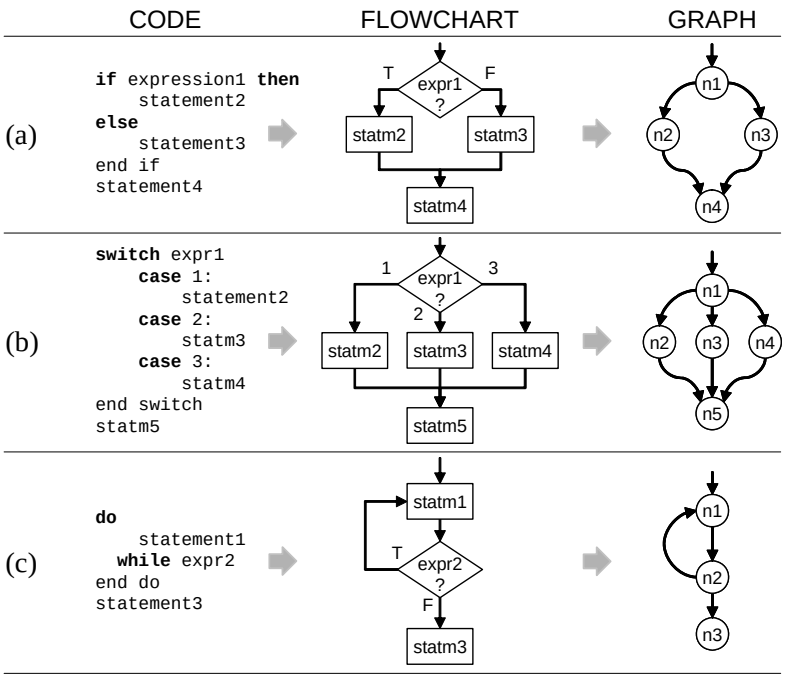


Figure 4-3: Converting software code into an abstract graph.

The cyclomatic complexity is additive. The complexity of several graphs considered as a group is equal to the sum of individual graphs' complexities.

There are two slightly different formulas for calculating cyclomatic complexity $V(G)$ of a graph G. The original formula by McCabe [1974] is

$$V(G) = e - n + 2 \cdot p \tag{4.5}$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components of the graph. Alternatively, [Henderson-Sellers & Tegarden, 1994] propose a *linearly-independent* cyclomatic complexity as

$$V_{LI}(G) = e - n + p + 1 \tag{4.6}$$

Because cyclomatic complexity metric is based on decisions and branches, which is consistent with the logic pattern of design and programming, it appeals to software professionals. But it is not without its drawbacks. Cyclomatic complexity ignores the complexity of sequential statements. In other words, any program with no conditional branching has zero cyclomatic complexity! Also, it does not distinguish different kinds of control flow complexity, such as loops vs. IF-THEN-ELSE statements or selection statements vs. nested IF-THEN-ELSE statements.

Cyclomatic complexity metric was originally designed to indicate a program's testability and understandability. It allows you to also determine the minimum number of unique tests that must be run to execute every executable statement in the program. One can expect programs with higher cyclomatic complexity to be more difficult to test and maintain, due to their higher complexity, and vice versa. To have good testability and maintainability, McCabe recommends that no program module should exceed a cyclomatic complexity of 10. Many software

refactorings are aimed at reducing the complexity of a program's conditional logic [Fowler, 2000; Kerievsky, 2005].

4.3 Measuring Module Cohesion

Cohesion is defined as a measure of relatedness or consistency in the functionality of a software unit. It is an attribute that identifies to which degree the parts within that unit belong together or are related to each other. In an object-oriented paradigm, a class can be a unit, the data can be attributes, and the methods can be parts. Modules with high cohesion are usually robust, reliable, reusable, and easy to understand while modules with low cohesion are associated with undesirable traits such as being difficult to understand, test, maintain, and reuse. Cohesion is an ordinal type of measurement and is usually expressed as "high cohesion" or "low cohesion."

We already encountered the term cohesion in Chapter 2, where it was argued that each unit of design, whether it is at the modular level or class level, should be focused on a single purpose. This means that it should have very few responsibilities that are logically related. Terms such as "intramodular functional relatedness" or "modular strength" have been used to address the notion of design cohesion.

4.3.1 Internal Cohesion or Syntactic Cohesion

Internal cohesion can best be understood as syntactic cohesion evaluated by examining the code of each individual module. It is thus closely related to the way in which large programs are modularized. Modularization can be accomplished for a variety of reasons and in a variety of ways.

A very crude modularization is to require that each module should not exceed certain size, e.g., 50 lines of code. This would arbitrarily quantize the program into blocks of about 50 lines each. Alternatively, we may require that each unit of design has certain prescribed size. For example, a package is required to have certain number of classes, or each class a certain number of attributes and operations. We may well end up with the unit of design or code which is performing unrelated tasks. Any cohesion here would be accidental or *coincidental cohesion*.

Coincidental cohesion does not usually occur in an initial design. However, as the design goes through multiple changes and modifications, e.g., due to requirements changes or bug fixes, and is under schedule pressures, the original design may evolve into a coincidental one. The original design may be patched to meet new requirements, or a related design may be adopted and modified instead of a fresh start. This will easily result in multiple unrelated elements in a design unit.

An Ordinal Scale for Cohesion Measurement

More reasonable design would have the contents of a module bear some relationship to each other. Different relationships can be created for the contents of each module. By identifying different types of module cohesion we can create a nominal scale for cohesion measurement. A stronger scale is an ordinal scale, which can be created by asking an expert to assess subjectively the quality of different types of module cohesion and create a rank-ordering. Here is an example ordinal scale for cohesion measurement:

Rank	Cohesion type	Quality
6	Functional cohesion	Good
5	Sequential cohesion	↓
4	Communication cohesion	↓
3	Procedural cohesion	↓
2	Temporal cohesion	↓
1	Logical cohesion	↓
0	Coincidental cohesion	Bad

Functional cohesion is judged to provide a tightest relationship because the design unit (module) performs a single well-defined function or achieves a single goal.

Sequential cohesion is judged as somewhat weaker, because the design unit performs more than one function, but these functions occur in an order prescribed by the specification, i.e., they are strongly related.

Communication cohesion is present when a design unit performs multiple functions, but all are targeted on the same data or the same sets of data. The data, however, is not organized in an object-oriented manner as a single type or structure.

Procedural cohesion is present when a design unit performs multiple functions that are procedurally related. The code in each module represents a single piece of functionality defining a control sequence of activities.

Temporal cohesion is present when a design unit performs more than one function, and they are related only by the fact that they must occur within the same time span. An example would be a design that combines all data initialization into one unit and performs all initialization at the same time even though it may be defined and utilized in other design units.

Logical cohesion is characteristic of a design unit that performs a series of similar functions. At first glance, logical cohesion seems to make sense in that the elements are related. However, the relationship is really quite weak. An example is the Java class `java.lang.Math`, which contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. Although all methods in this class are logically related in that they perform mathematical operations, they are entirely independent of each other.

Ideally, object-oriented design units (classes) should exhibit the top two types of cohesion (functional or sequential), where operations work on the attributes that are common for the class.

A serious issue with this cohesion measure is that the success of any module in attaining high-level cohesion relies purely on human assessment.

Interval Scales for Cohesion Measurement

We are mainly interested in the cohesion of object-oriented units of software, such as classes. **Class cohesion** captures relatedness between various members of a class: attributes and operations (or, methods). Class cohesion metrics can be broadly classified into two groups:

1. **Interface-based metrics** compute class cohesion from information in method signatures
2. **Code-based metrics** compute class cohesion in terms of attribute accesses by methods

We can further classify code-based cohesion metrics into four sub-types based on the methods of quantification of cohesion:

2.a) Disjoint component-based metrics count the number of disjoint sets of methods or attributes in a given class.

2.b) Pairwise connection-based metrics compute cohesion as a function of number of connected and disjoint method pairs.

2.c) Connection magnitude-based metrics count the accessing methods per attribute and indirectly find an attribute-sharing index in terms of the count (instead of computing direct attribute-sharing between methods).

2.d) Decomposition-based metrics compute cohesion in terms of recursive decompositions of a given class. The decompositions are generated by removal of pivotal elements that keep the class connected.

These metrics compute class cohesion using manipulations of class elements. The key elements of a class C are its a attributes $A_1, \ldots, A_a$, m methods $M_1, \ldots, M_m$, and the list of p parameter (or, argument) types of the methods $P_1, \ldots, P_m$. The following sections describe various approaches to computing class cohesion.

Many existing metrics qualify the class as either “cohesive” or “not cohesive,” and do not capture varying strengths of cohesion. However, this approach makes it hard to compare two cohesive or two non cohesive classes, or to know whether a code modification increased or reduced the degree of cohesiveness. If one wishes to compare the cohesion of two different versions of software, it is necessary to use a metric that can calculate not just whether a module is cohesive or not cohesive but also the degree of its cohesiveness. Assuming that both the versions of our software are cohesive, this would enable us to judge which version is better designed and more cohesive.

4.3.2 Interface-based Cohesion Metrics

Interface-based cohesion metrics are design metrics that help evaluate cohesion among methods of a class early in the analysis and the design phase. These metrics evaluate the consistency of methods in a class’s interface using the lists of parameters of the methods. They can be applied on class declarations that only contain method prototypes (method types and parameter types) and do not require the class implementation code. One such metric is **Cohesion Among Methods of Classes** (CAMC). The CAMC metric is based on the assumption that the parameters of a method reasonably define the types of interaction that method may implement.

Figure 4-4

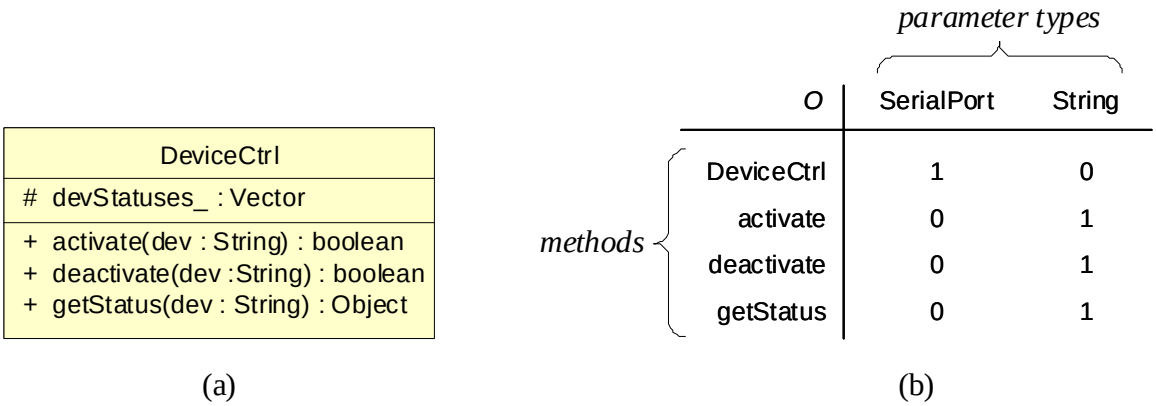


Figure 4-4: Class (a) and its parameter occurrence matrix (b).

To compute the CAMC metric value, we determine a union of all parameters of all the methods of a class. A set M_i of parameter object types for each method is also determined. An intersection (set P_i) of M_i with the union set T is computed for all methods in the class. A ratio of the size of the intersection (P_i) set to the size of the union set (T) is computed for all methods. The summation of all intersection sets P_i is divided by product of the number of methods and the size of the union set T, to give a value for the CAMC metric. Formally, the metric is

$$CAMC(C) = \frac{1}{kl}\sum_{i=1}^{k}\sum_{j=1}^{l} o_{ij} = \frac{\sigma}{kl} \tag{4.7}$$

4.3.3 Cohesion Metrics using Disjoint Sets of Elements

An early metric of this type is the *Lack of Cohesion of Methods* (LCOM1). This metric counts the number of pairs of methods that do not share their class attributes. It considers how many disjoint sets are formed by the intersection of the sets of the class attributes used by each method. Under LCOM1, the perfect cohesion achieved when all methods access all attributes. Because of perfect cohesion, we expect the lack-of-cohesion value to be 0. At the opposite end of the spectrum, each method accesses only a single attribute (assuming that $m = a$). In this case, we expect LCOM = 1, which indicates extreme lack of cohesion.

A formal definition of LCOM1 follows. Consider a set of methods $\{M_i\}$ (i = 1, …, m} accessing a set of attributes $\{A_j\}$ (j = 1, …, a}. Let the number of attributes accessed by each method, M_i, be denoted as $\alpha(M_i)$ and the number of methods which access each attribute be $\mu(A_j)$. Then the lack of cohesion of methods for a class C_i is given formally as

$$\text{LCOM1}(C_i) = \frac{m - \left(\frac{1}{a}\cdot\sum_{j=1}^{a}\mu(A_j)\right)}{m-1} \tag{4.8}$$

This version of LCOM is labeled as LCOM1 to allow for subsequent variations, LCOM2, LCOM3, and LCOM4. Class cohesion, LCOM3, is measured as the number of connected

components in the graph. LCOM2 calculates the difference between the number of method pairs that do or do not share their class attributes. LCOM2 is classified as a Pairwise Connection-Based metric (Section ____). See the bibliographical notes (Section 4.7) for references on LCOM metrics.

4.3.4 Semantic Cohesion

Cohesion or module "strength" refers to the notion of a module level "togetherness" viewed at the system abstraction level. Thus, although in a sense it can be regarded as a system design concept, we can more properly regard cohesion as a semantic concern expressed *of* a module evaluated externally to the module.

Semantic cohesion is an externally discernable concept that assesses whether the abstraction represented by the module (class in object-oriented approach) can be considered to be a "whole" semantically. Semantic complexity metrics evaluate whether an individual class is really an abstract data type in the sense of being complete and also coherent. That is, to be semantically cohesive, a class should contain everything that one would expect a class with those responsibilities to possess but no more.

It is possible to have a class with high internal, syntactic cohesion but little semantic cohesion. Individually semantically cohesive classes may be merged to give an externally semantically nonsensical class while retaining internal syntactic cohesion. For example, imagine a class that includes features of both a person and the car the person owns. Let us assume that each person can own only one car and that each car can only be owned by one person (a one-to-one association). Then *person_id* ↔ *car_id*, which would be equivalent to data normalization. However, classes have not only data but operations to perform various actions. They provide behavior patterns for (1) the person aspect and (2) the car aspect of our proposed class. Assuming no intersecting behavior between PERSON and CAR, then what is the meaning of our class, presumably named CAR_PERSON? Such a class could be internally highly cohesive, yet semantically *as a whole class seen from outside* the notion expressed (here of the thing known as a person-car) is nonsensical.

4.4 Coupling

Coupling metrics are a measure of how interdependent different modules are of each other. High coupling occurs when one module modifies or relies on the internal workings of another module. Low coupling occurs when there is no communication at all between different modules in a program. Coupling is contrasted with cohesion. Both cohesion and coupling are ordinal measurements and are defined as "high" or "low." It is most desirable to achieve low coupling and high cohesion.

Tightly coupled vs. loosely coupled

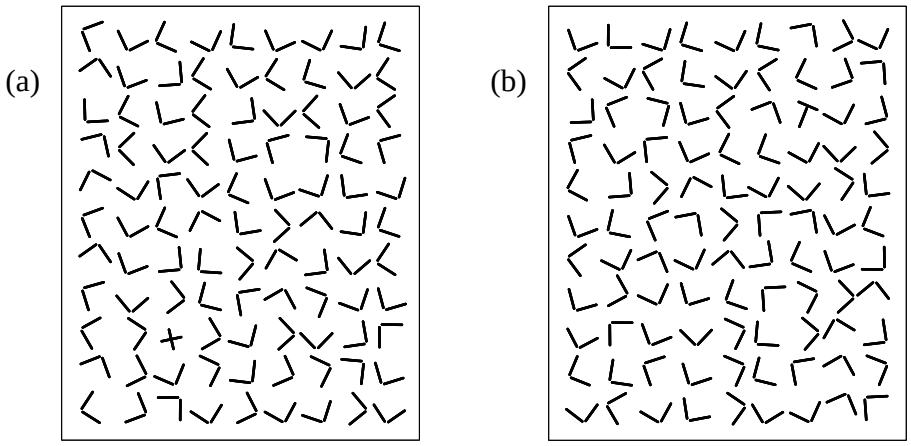


Figure 4-5: Random arrays illustrate information complexity vs. depth. See text for details.

4.5 Psychological Complexity

"Then he explained that what can be observed is really determined by the theory. He said, you cannot first know what can be observed, but you must first know a theory, or produce a theory, and then you can define what can be observed." —Heisenberg's recollection of his first meeting with Einstein

"I have had my results for a long time: but I do not yet know how I am to arrive at them."
—Karl Friedrich Gauss

One frustration with software complexity measurement is that, unlike placing a physical object on a scale and measuring its weight, we cannot put a software object on a "complexity scale" and read out the amount. Complexity seems to be an interpreted measure, much like person's health condition and it has to be stated as an "average case." Your doctor can precisely measure your blood pressure, but a specific number does not necessarily correspond to good or bad health. The doctor will also measure your heart rate, body temperature, and perhaps several other parameters, before making an assessment about your health condition. Even so, the assessment will be the best guess, merely stating that on average such and such combination of physiological measurements corresponds to a certain health condition. Perhaps we should define software object complexity similarly: as a statistical inference based on a set of directly measurable variables.

4.5.1 Algorithmic Information Content

I already mentioned that our abstractions are unavoidably approximate. The term often used is "coarse graining," which means that we are blurring detail in the world picture and single out only the phenomena we believe are relevant to the problem at hand. Hence, when defining complexity it is always necessary to specify the level of detail up to which the system is described, with finer details being ignored.

One way of defining the complexity of a program or system is by means of its description, that is, the length of the description. I discussed above the merits of using size metrics as a complexity

measure. Some problems mentioned above include: size could be measured differently; it depends on the language in which the program code (or any other accurate description of it) is written; and, the program description can be unnecessarily stuffed to make it appear complex. A way out is to ignore the language issue and define complexity in terms of the description length.

Suppose that two persons wish to communicate a system description at distance. Assume they are employing language, knowledge, and understanding that both parties share (and know they share) beforehand. The *crude complexity* of the system can be defined as the *length of the shortest message* that one party needs to employ to describe the system, at a given level of coarse graining, to the distant party.

A well-known such measure is called *algorithmic information content*, which was introduced in 1960s independently by Andrei N. Kolmogorov, Gregory Chaitin, and Ray Solomonoff. Assume an idealized general-purpose computer with an infinite storage capacity. Consider a particular message string, such as "`aaaaabbbbbbbbbb`." We want to know: what is the shortest possible program that will print out that string and then stop computing? **Algorithmic information content** (AIC) is defined as the length of the *shortest* possible program that prints out a given string. For the example string, the program may look something like: `P a{5}b{10}`, which means "Print 'a' five times and 'b' ten times."

Information Theory

Logical Depth and Crypticity

"...I think it better to write a long letter than incur loss of time..." —Cicero

"I apologize that this letter is so long. I did not have the time to make it short." —Mark Twain

"If I had more time, I would have written a shorter letter."
—variously attributed to Cicero, Pascal, Voltaire, Mark Twain, George Bernard Shaw, and T.S. Elliot

"The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich may find hard to pay." —C.A.R. Hoare

I already mentioned that algorithmic information content (AIC) does not exactly correspond to everyday notion of complexity because under AIC random strings appear as most complex. But there are other aspects to consider, as well. Consider the following description: "letter X in a random array of letters L." Then the description "letter T in a random array of letters L" should have about the same AIC. Figure 4-5 pictures both descriptions in the manner pioneered by my favorite teacher Bela Julesz. If you look at both arrays, I bet that you will be able to quickly notice X in Figure 4-5(a), but you will spend quite some time scanning Figure 4-5(b) to detect the T! There is no reason to believe that human visual system developed a special mechanism to recognize the pattern in Figure 4-5(a), but failed to do so for the pattern in Figure 4-5(b). More likely, the same general pattern recognition mechanism operates in both cases, but with much less success on Figure 4-5(b). Therefore, it appears that there is something missing in the AIC notion

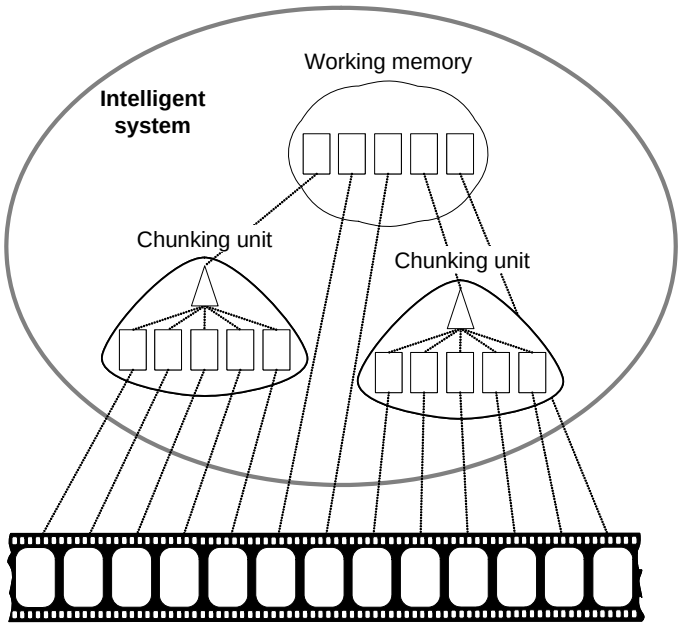


Figure 4-6: A model of a limited working memory.

of complexity—an apparently complex description has low AIC. The solution is to include the computation time.

Charles Bennett defined *logical depth* of a description to characterize the difficulty of going from the shortest program that can print the description to the actual description of the system. Consider not just the shortest program to print out the string, but a set of short programs that have the same effect. For each of these programs, determine the length of time (or number of steps) needed to compute and print the string. Finally, average the running times so that shorter programs are given greater weight.

4.6 Effort Estimation

"Adding manpower to a late software project makes it later."
—Frederick P. Brooks, Jr., *The Mythical Man-Month*

"A carelessly planned project will take only twice as long."
—The law of computerdom according to Golub

"The first 90 percent of the tasks takes 10 percent of the time and the last 10 percent takes the other 90 percent." —The ninety-ninety rule of project schedules

4.6.1 Deriving Project Duration from Use Case Points

Use case points (*UCP*) are a measure of software size (Section 4.2.1). We can use equation (1.2) given in Section 1.2.5 to derive the project duration. For this purpose we need to know the team's *velocity*, which represents the team's rate of progress through the use cases (or, the team's productivity). Here is the equation that is equivalent to equation (1.2), but using a Productivity Factor (*PF*):

$$Duration = UCP \times PF \tag{4.9}$$

The Productivity Factor is the ratio of development person-hours needed per use case point. Experience and statistical data collected from past projects provide the data to estimate the initial *PF*. For instance, if a past project with a *UCP* of 112 took 2,550 hours to complete, divide 2,550 by 112 to obtain a *PF* of 23 person-hours per use case point.

If no historical data has been collected, the developer can consider one of these options:

1. Establish a baseline by computing the *UCP* for projects previously completed by your team (if such are available).
2. Use a value for *PF* between 15 and 30 depending on the development team's overall experience and past accomplishments (Do they normally finish on time? Under budget? etc.). For a team of beginners, such as undergraduate students, use the highest value (i.e., 30) on the first project.

A different approach was proposed by Schneider and Winters [2001]. Recall that the environmental factors (Table 4-7) measure the experience level of the people on your project and the stability of your project. Any negatives in this area mean that you will have to spend time training people or fixing problems due to instability (of requirements). The more negatives you have, the more time you will spend fixing problems and training people and less time you will have to devote to your project.

Schneider and Winters suggested counting the number of environmental factors among E1 through E6 (Table 4-8) that have the perceived impact less than 3 and those among E7 and E8 with the impact greater than 3. If the total count is 2 or less, assume 20 hours per use case point. If the total is 3 or 4, assume 28 hours per use case. Any total greater than 4 indicates that there are too many environmental factors stacked against the project. The project should be put on hold until some environmental factors can be improved.

Probably the best solution for estimating the Productivity Factor is to calculate your organization's own historical average from past projects. This is why collecting historic data is important for improving effort estimation on future projects. After a project completes, divide the number of actual hours it took to complete the project by the *UCP* number. The result becomes the new *PF* that can be used in the future projects.

When estimating the duration in calendar time, is important to avoid assuming ideal working conditions. The estimate should account for corporate overhead—answering email, attending meetings, and so on. Suppose our past experience suggests a *PF* of 23 person-hours per use case point and our current project has 94 use case points (as determined Section 4.2.1). Equation (4.9)

gives the duration as $94 \times 23 = 2162$ person-hours. Obviously, this does not imply that the project will be completed in $2162 / 24 \approx 90$ days! A reasonable assumption is that each developer will spend about 30 hours per week on project tasks and the rest of their time will be taken by corporate overhead. With a team of four developers, this means the team will make $4 \times 30 = 120$ hours per week. Dividing 2162 person-hours by 120 hours per week we obtain a total of approximately 18 weeks to complete this project.

4.7 Summary and Bibliographical Notes

In this chapter I described two kinds of software measurements. One kind works with scarce artifacts that are available early on in a project, such as customer statement of requirements for the planned system. There is a major subjective component to these measurements, and it works mainly based on guessing and past experience with similar projects. The purpose of this kind of measurements is to *estimate the project duration* and cost of the effort, so to negotiate the terms of the contract with the customer who is sponsoring the project.

The other kind of software measurements works with actual software artifacts, such as UML designs or source code. It aims to measure intrinsic properties of the software and avoid developer's subjective guesses. Because it requires that the measured artifacts already exist in a completed or nearly completed condition, it cannot be applied early on in a project. The purpose of this kind of measurements is to *evaluate the product quality*. It can serve as a test of whether the product is ready for deployment, or to provide feedback to the development team about the potential weaknesses that need to be addressed.

An early project effort estimate helps managers, developers, and testers plan for the resources a project requires. The *use case points* (UCP) method has emerged as one such method. It is a mixture of intrinsic software properties, measured by Unadjusted Use Case Points (UUCP) as well as technical (TCF) and environmental factors (ECF), which depend on developer's subjective estimates. The UCP method quantifies these subjective factors into equation variables that can be adjusted over time to produce more precise estimates. Industrial case studies indicate that the UCP method can produce an early estimate within 20% of the actual effort.

Section 4.2: What to Measure?

[Henderson-Sellers, 1996] provides a condensed review of software metrics up to the publication date, so it is somewhat outdated. It is technical and focuses on metrics of structural complexity.

Horst Zuse, History of Software Measurement, Technische Universität Berlin, Online at: http://irb.cs.tu-berlin.de/~zuse/sme.html

[Halstead, 1977] distinguishes software science from computer science. The premise of software science is that any programming task consists of selecting and arranging a finite number of program "tokens," which are basic syntactic units distinguishable by a compiler: operators and operands. He defined several software metrics based on these tokens. However, software science

has been controversial since its introduction and has been criticized from many fronts. Halstead's work has mainly historical importance for software measurement because it was instrumental in making metrics studies an issue among computer scientists.

Use case points (UCP) were first described by Gustav Karner [1993], but his initial work on the subject is closely guarded by Rational Software, Inc. Hence, the primary sources describing Karner's work are [Schneider & Winters, 2001] and [Ribu, 2001]. UCP was inspired by Allan Albrecht's "Function Point Analysis" [Albrecht, 1979]. The weighted values and constraining constants were initially based on Albrecht, but subsequently modified by people at Objective Systems, LLC, based on their experience with Objectory—a methodology created by Ivar Jacobson for developing object-oriented applications.

My main sources for use case points were [Schneider & Winters, 2001; Ribu, 2001; Cohn, 2005]. [Kusumoto, et al., 2004] describes the rules for a system that automatically computes the total UCP for given use cases. I believe these rules are very useful for a beginner human when computing UCPs for a project.

Many industrial case studies verified the estimation accuracy of the UCP method. These case studies found that the UCP method can produce an early estimate within 20% of the actual effort, and often closer to the actual effort than experts or other estimation methodologies. Mohagheghi et al. [2005] described the UCP estimate of an incremental, large-scale development project that was within 17% of the actual effort. Carroll [2005] described a case study over a period of five years and across more than 200 projects. After applying the process across hundreds of sizable software projects (60 person-months average), they achieved estimating accuracy of less than 9% deviation from actual to estimated cost on 95% of the studied projects. To achieve greater accuracy, Carroll's estimation method includes a risk coefficient in the UCP equation.

Section 4.3: Measuring Module Cohesion

The ordinal scale for cohesion measurement with seven levels of cohesion was proposed by Yourdon and Constantine [1979].

[Constantine *et al*., 1974; Eder *et al*., 1992; Allen & Khoshgoftaar, 1999; Henry & Gotterbarn, 1996; Mitchell & Power, 2005]

See also: http://c2.com/cgi/wiki?CouplingAndCohesion

B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion: Towards a valid suite of object-oriented metrics," *Object-Oriented Systems*, vol. 3, no. 3, 143-158, 1996.

[Joshi & Joshi, 2010; Al Dallal, 2011] investigated the discriminative power of object-oriented class cohesion metrics.

Section 4.4: Coupling

Section 4.5: Psychological Complexity

[Bennett, 1986; 1987; 1990] discusses definition of complexity for physical systems and defines logical depth.

Section 4.6: Effort Estimation

Problems

Problem 4.1

Problem 4.2

Problem 4.3

(CYCLOMATIC/MCCABE COMPLEXITY) Consider the following quicksort sorting algorithm:

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

where the PARTITION procedure is as follows:

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4     do if A[j] ≤ x
5           then i ← i + 1
6                exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

(a) Draw the flowchart of the above algorithm.
(b) Draw the corresponding graph and label the nodes as n_1, n_2, … and edges as e_1, e_2, …
(c) Calculate the cyclomatic complexity of the above algorithm.

Problem 4.4