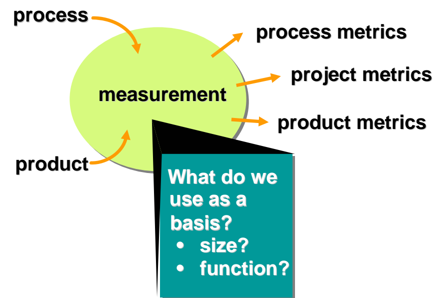
Estimating Software Size and Object Oriented Metrics

Sources:

1. Roger S. Pressman, Software Engineering – A Practitioner's Approach, 5th Edition, ISBN 0-07-365578-3, McGraw-Hill, 2001 (Chapters 4 & 24)
2. Stephen H. Kan, Metrics and Models in Software Quality Engineering, 2nd Edition, ISBN 0-201-72915-6, Addison-Wesley, 2003
3. Shyam Chidamber and Chris Kemerer, "A Metrics Suite for Object Oriented Design," IEEE TOSE 20:6 June 1994, 476-493
4. Rachel Harrison, Steve Counsell, and Reuben Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," IEEE TOSE 24:6 June 1998, 491-496

A Good Manager Measures



Why do we Measure?

- To characterize – **to gain understanding of process, products, resources, and environments, and to establish baselines for future assessments**
- To evaluate – **to determine status with respect to plans.**
- To predict – **so that we can plan.**
- To improve – **we gather information to help us identify road blocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance.**

Product Metrics

- focus on the quality of deliverables
- measures of analysis model
- complexity of the design
  - internal algorithmic complexity
  - architectural complexity
  - data flow complexity
- code measures (e.g., Halstead)
- measures of process effectiveness
  - e.g., defect removal efficiency

UML-Based Sizing and Use Case Points

- One approach to software sizing is to use the products of analysis as a basis for early reliable measures of size. This is called "process estimation.)
- Though requirements are still a crude way to quantify the functionality of a software product, UML provides a standard notation, and it is considered an industry standard.
- A use case is one of the components of UML. Use cases do not capture nonfunctional requirements nor are they simply a functional decomposition of the system.
- Use cases capture functions and data in user terms.
- Use case points (UCPs) are metric to estimate effort for projects [Gustav Karner, 1993].

Steps to Count Use Case Points

1. Identify the actors and assign a weight based on how they interact with the system of interest:

Actor Type	Description of Interface	Weight
Simple	Another system via a defined application programming interface (API)	1
Average	Another system via a protocol, or a person via a text-based terminal	2
Complex	A person interacting via a graphical user interface (GUI)	3

2. Sum the weights for the actors in all use cases to obtain the unadjusted actor weight, UAW

3. Identify use cases, and assign a complexity to each use case based on the number of transactions or scenarios that each use case contains:

Complexity	*# of Transactions*	*Weight*
Simple	1–3	5
Average	4–7	10
Complex	8 or more	15

Alternatively, you can use “analysis classes” to estimate the use case complexity. I prefer this method because it less subjective.

4. Sum the weight for all the use cases to obtain the unadjusted use weight, UUCW.
5. Sum UAW and UUCW to obtain the size in unadjusted use case points (UUCPs).

6. Adjust for the technical complexity of the product by rating the degree of influence of each of the 13 factors. The ratings range from 0 to 5; 0 means that the factor is irrelevant for the project; 5 means that it is essential. The 13 factors and their associated weights are:

Factor	*Description*	*Weight*
T1	Distributed system	2
T2	Response or throughput performance objectives	2
T3	End-user efficiency	1
T4	Complex internal processing	1
T5	Reusable code	1
T6	Easy to install	0.5
T7	Easy to use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent	1
T11	Includes security features	1
T12	Provides access for third parties	1
T13	Special user training facilities are required	1

7. For each factor, multiply the degree of influence by the weight, and sum the products to obtain the technology sum, TSUM. Use the weights table shown in the table in the preceding step.
8. Compute the technical complexity factor, TCF, using TCF = 0.6 + 0.01 * TSUM
9. Adjust for the “envirenment,” which addresses the skills and training of the staff, precedentedness, and requirements stability. There are eight factors:

Factor	*Description*	*Weight*
F1	Familiar with Rational Unified Process	1.5
F2	Application experience	0.5
F3	Object-oriented experience	1
F4	Lead analyst capability	0.5
F5	Motivation	1
F6	Stable requirements	2
F7	Part-time workers	–1
F8	Difficult programming language	–1

10. Rate each factor’s influence from 0 to 5, with 3 denoting “average.” For factors F1 through F4, 0 means no experience in that area, and 5 means expert. For factor F5, 0 means no motivation, and 5 means high motivation. For factor 6, 0 means extremely unstable requirements and 5 means unchanging requirements. For factor F7, 0 means no part-time staff, and 5 means all part-time staff. For factor F8, 0 means an easy-to-use programming language, and 5 means a very difficult programming language.

11. For each factor, multiply the degree of influence by the weight, and sum the products to obtain the environment sum ESUM

12. Compute the environment factor, EF, using EF = 1.4 – 0.03*ESUM

13. Compute the size in (adjusted) Use Case Points (UCPs) using UCP = UUCP * TCF * EF

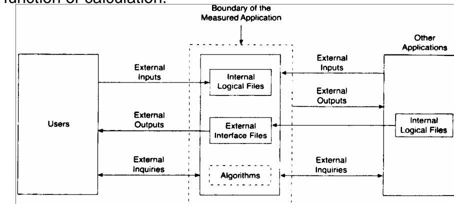
Function Points

- Function points [Allan J. Albrecht, 1979] are a software size measure designed to meet three goals:

- Gauge delivered functionality in terms users can understand
- Be independent of technique, technology, and programming language
- Give a reliable indication of software size during early design

The Counting Rules

- Function point analysis (FPA) quantifies product functionality in terms of five externally visible system elements, called *function types*, that are readily understood by both users and developers:
- **EI: External input** – is a related group of user data or control information that enters the boundary of the application and adds or changes data in an internal logical file, or used to perform some function or calculation.
-



- **EO: External output** – is a related group of user data or control information that leaves the boundary of the application.
- **EQ: External query** – is a related group of user data or control information that enters the boundary of the application and generates an immediate output of a related group of user data or control information. A query is a set of selection criteria that are used to extract information from an existing database.
- **ILF: Internal logical file** – is a user identifiable group of logically related data or control information that (i) resides within the boundary of the application, and (ii) is maintained and used by the application.
- **EIF: External interface file** – is a user-identifiable group of logically related data or control information that (i) resides outside of the application boundary, and (ii) is used by the application for some of its processing.
- Function point counting is a type of Linear Method.

Weights for Function Points and Feature Points

- Feature points [Capers Jones, 1985] is a simplified size measure compared to function points.
- Feature point also identifies a sixth data type, algorithms, to account for complex calculation.

Function Types	*IFPUG Function Points (1984)*			*SPR Feature Points (1985)*
	Low	*Avgerage*	*High*	
Input	3	4	6	4
Output	4	5	7	5
Inquiry	3	4	6	4
Internal logical files	4	10	15	7
External interface files	5	7	10	7
Algorithm	None			3

- For each function type and complexity, the count is multiplied by the corresponding weight.
- Summing the results for all 15 pairs (5 function types and 3 complexity levels) gives the total number of unadjusted function points (UFPs) for the software product.
- This total (UFPs) is adjusted for the general system characteristics (GSCs) of the product.
- Table (next slide) lists the 14 GSCs and some of the main factors considered in assigning a value.
- Each factor is rated based on its "degree of influence" (0 = no influence, up to 5 = strong influence).
- Summing these 14 ratings gives the total degree of influence (TD1) which is used to compute value adjustment factor (VAF), as follows:
- $TDI = \sum^{14}_{j=1} Rating$ and $VAF = 0.65 + 0.01*TDI$

- Applying the value adjustment factor gives the software size in adjusted function points (AFPs): Size(AFPs) = VAF*Size(UFP)
- The VAF can increase or decrease the size by no more than 35%.
- The "dynamic range" of the size adjustment is 2 (=1.35/0.65)

Characteristic	*Considerations*
Data communications	Batch versus interactive
Distributed data processing	Client/server, real-time process control
Performance	Response time, throughput
Heavily used configuration	Security or timing considerations
Transaction rate	Batch, online, real time
Online data entry	Interactive data entry and control
End-user efficiency	User interface, multilingual support
Online update	Updates to internal logical files
Complex processing	Control logic, numeric calculations
Reusability	Parameterized to permit customization by user
Installation ease	Conversion of operational data
Operational ease	Startup, backup, recovery, unattended operation
Multiple sites	Differences in hardware and software environments
Facilitate change	Ad hoc queries, table-driven logic

Advantages and Disadvantages of Function-Based Sizing

- It facilitates the dialogue between the user of the system and the developer.
- FP estimates are generally claimed to be accurate within ±10%
- FP counting cannot take place until the requirements are reasonably well understood and the high level design of the system is known.
- A main disadvantage of FPs is that they must be counted manually. This is expensive.
- FP only gives the (functional) size of a software not the estimate development effort or time.
- Other concerns with functional size measurement [Norman Fenton and Shari Pfleeger] are given in the table below.

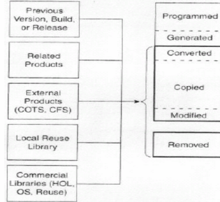
Some Concerns with Function Points

Element Definitions
• The definitions need clarification and interpretation for new types of software. • Analysts sometimes fail to classify inputs, outputs, queries, and "logical files" consistently even with detailed rules.
Element Weights for Unadjusted Function Point
• The weights are subjective, and based on Albrecht's experience. They may not be appropriate for other product types and development environments. • The weights may be different for "value" and for "development effort."
General System Characteristics (GSC)
• The Likert ratings are subjective. • They mix user-perceived complexity and internal system complexity. (It might be better to separate these aspects.)
Value Adjustment Factor (VAF) and Adjusted Function Points (AFP)
• Assigning all ratings as Average (= 3) gives VAF = 1.07, not 1.0, as you expect. • Using VAF does not significantly improve the accuracy of the estimated effort. • The calculation of AFP mixes measurements from ratio and ordinal scales.
Process Issues
• You cannot count function points very early. (You must have a high-level design.) • Initial counts are sometimes much less than the final counts of the delivered product.

Defining Source Lines of Code (SLOC)

- Code may originate from different sources as shown below and this must be considered when counting SLOC

Programmed = New, custom, built from scratch, developed, developmental
Generated = Produced by a tool from high-level specifications or designs
Converted = (Automatically) translated
Copied = (Re)used "as is," (reused) unmodified, reused verbatim
Modified = Adapted, leveraged
Removed = Deleted



The following table provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

Programming Language	LOC/FP (average)
Assembly language	**320**
C	**128**
COBOL	**106**
FORTRAN	**106**
Pascal	**90**
C++	**64**
Ada95	**53**
Visual Basic	**32**
Smalltalk	**22**
Powerbuilder (code generator)	**16**
SQL	**12**

The Constructive Cost Model (COCOMO

- **COCOMO is the classic LOC cost-estimation formula.**
- **It was created by Barry Boehm in the 1970s**
- **He used thousand delivered source instructions (KDSI) as his unit of size. KLOC is equivalent.**
- **His unit of effort is the programmer-month (PM).**
- **Boehm divided the historical project data into three types of projects:**
  1. **Application (separate, organic, e.g., data processing, scientific)**
  2. **Utility programs (semidetached, e.g., compilers, linkers, analyzers)**
  3. **System programs (embedded)**

- **He determined the values of the parameters for the cost model for determining effort:**

1. **Application programs: $PM = 2.4*(KDSI)^{1.05}$**
2. **Utility programs: $PM = 3.0*(KDSI)^{1.12}$**
3. **Systems programs: $PM = 3.6*(KDSI)^{1.20}$**

- **Boehm also determined development time (TDEV) in programmer-months:**

1. **Application programs: $TDEV = 2.5*(PM)^{0.38}$**
2. **Utility programs: $TDEV = 2.5*(PM)^{0.35}$**
3. **Systems programs: $TDEV = 2.5*(PM)^{0.32}$**

Conventional Methods: LOC/FP Approach

The CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter.

- **Compute LOC/FP using estimates of information domain values**
- **Use historical effort for the project**
- **For our purposes, we assume that further refinement has occurred and that the following major software functions are identified:**
- **User interface and control facilities (UICF)**
- **Two-dimensional geometric analysis (2DGA)**
- **Three-dimensional geometric analysis (3DGA)**
- **Database management (DBM)**
- **Computer graphics display facilities (CGDF)**
- **Peripheral control function (PCF)**
- **Design analysis modules (DAM)**

Example: LOC Approach

Functions	estimated LOC	LOC/pm	$/LOC	Cost	Effort (months)
UICF	2340	315	14	32,000	7.4
2DGA	5380	220	20	107,000	24.4
3DGA	6800	220	20	136,000	30.9
DSM	3350	240	18	60,000	13.9
CGDF	4950	200	22	109,000	24.7
PCF	2140	140	28	60,000	15.2
DAM	8400	300	18	151,000	28.0
Totals	33,360			655,000	145.0

Example: FP Approach

measurement parameter	count		weight		
number of user inputs	40	x	4	=	160
number of user outputs	25	x	5	=	125
number of user inquiries	12	x	4	=	48
number of files	4	x	7	=	28
number of ext.interfaces	4	x	7	=	28
algorithms	60	x	3	=	180
count-total					569
complexity multiplier					.84
feature points					478

X 0.25 p-m / FP = 120

Process-Based Estimation

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function ↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

Object Oriented Measurement

- **The measurement of object-oriented software has the same goals as the measurement of conventional software – trying to understand the characteristics of the software.**
- **Traditional software measurement uses control flow graph as the basic abstraction of the software. The control flow graph does not appear to be useful as an abstraction of object-oriented software.**
- **The intuitive problem with applying the traditional software metrics is that the complexity of object-oriented software does not appear to be in the control structure.**
- **In most object-oriented software, the complexity appears to be in the calling patterns among the methods.**

Weighted Methods per Class (WMC)

- **The WMC metric is based on the intuition that the number of methods per class is a significant indication of the complexity of the software.**
- **Let C be a set of classes each with the number of methods $M_1, ..., M_n$. Let $c_1, ..., c_n$ be the complexity (weights) of the classes (assume c_i is equal to 1).**

$$\mathrm{WMC} = \frac{1}{n} * \sum_{i=0}^{n} c_i * M_i$$

- **This is the only metric that is averaged over the classes in a system**

Depth of Inheritance Tree (DIT)

- **The depth of inheritance tree metric is just the maximum length from any node to the root of the inheritance tree for that class.**
- **Inheritance can add to complexity of software. This metric is calculated for each class.**

Number of Children (NOC)

- **Not only is the depth of the inheritance tree significant, but the width of the inheritance tree.**
- **The number of children metric is the number of immediate sub-classes subordinated to a class in the inheritance hierarchy. This metric is calculated for each class.**

Coupling between Object Classes (CBO)

- **In object-oriented, coupling can be defined as the use of methods or attributes in another class.**
- **Two classes will be considered coupled when methods declared in one class use methods or instance variables defined by the other class.**
- **Coupling is symmetric. If class A is coupled to class B, then B is coupled to A.**
- **The coupling between object classes (CBO) metric will be the count of the number of other classes to which it is coupled.**
- **This metric is calculated for each class.**

Response For a Class (RFC)

- **The response set of a class, {RS}, is the set of methods that can potentially be executed in response to a message received by an object of that class.**
- **It is the union of all methods in the class and all methods called by methods in the class.**
- **It is only counted on one level of call.**

RFC = |RS|

- **The metric is calculated for each class**

Lack of Cohesion in Methods (LCOM)

- **A module (or class) is cohesive if everything is closely related. The lack of cohesion in methods metric tries to measure the lack of cohesiveness.**
  Let I_i be the set of instance variables used by method I.
  Let P be set of pairwise null intersections of I_i.
  Let Q be set of pairwise nonnull intersections.
- **LCOM metric can be visualize by considering a bipartite graph. One set of nodes consists of attributes, and the other set of nodes consists of the functions. An attribute is linked to a function if that function accesses or sets that attributes.**
- **The set of arcs is the set Q. If there are n attributes and m functions, then there are a possible $n * m$ arcs. So, the size of P is $n + m$ minus the size of Q.**

$$\text{LCOM} = \max(|P| - |Q|, 0)$$

- **This metric is calculated on a class basis.**

The MOOD (Metrics for Object Oriented Design)

The MOOD suite of metrics is intended as a complete set that measures the attributes of encapsulation, inheritance, coupling, and polymorphism of a system.

- **Let TC be the total number of classes in the system.**
- **Let $M_d(C_i)$ be the number of methods declared in a class i.**
- **Consider the predicate $Is_visible(M_{m,i}, C_j)$, where $M_{m,i}$ is the method m in class i and C_j is the class j. This predicate is 1 if i != j and C_j may call $M_{m,i}$. Otherwise, the predicate is 0. For example, a pubic method in C++ is visible to all other classes. A private method in C++ is not visible to other classes.**

The visibility, $V(M_{m,i})$, of a method, $M_{m,i}$ is defined as follows:

$$V(M_{m,i}) = \frac{\sum_{j=1}^{TC} \text{Is_visible}(M_{m,i}, C_j)}{TC - 1}$$

Encapsulation – MHF and AHF

- **The *method hiding factor* (MHF) and the *attribute hiding factor* (AHF) attempt to measure the encapsulation.**

$$\text{MHF} = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{m,i})}{\sum_{i=1}^{TC} M_d(C_i)}$$

$$\text{AHF} = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{m,i})}{\sum_{i=1}^{TC} A_d(C_i)}$$

- **Where $A_d(C_i)$ is the number of attributes declared in a class and**
- **$V(A_{m,i})$ is the visibility of an attribute, $A_{m,i}$**

Example 1- Calculate MHF and AHF for the C++ code below:

```
Class A{
   int a;
public:
   void x();
   void y();
};
Class B {
   int b;
   int bb;
   void w();
public:
   void z():
};
Class C {
   int c;
   void v();
};
TC = 3
```

method	is_vis(A)	is_vis(B)	is_vis(C)	$V(M_{m,i})$
A::x()	0	1	1	1
A::y()	0	1	1	1
B::w()	0	0	0	0
B::z()	1	0	1	1
C::v()	0	0	0	0

MHF = 2/5 = 0.4

attribute	is_vis(A)	is_vis(B)	is_vis(C)	$V(A_{m,i})$
A::a()	0	0	0	0
B::b()	0	0	0	0
B::bb()	0	0	0	0
C::c()	0	0	0	0

AHF = 4/4 = 1.0

Inheritance Factor – MIF and AIF

- **There are two measures of the inheritance, the method inheritance factor (MIF) and the attribute inheritance factor (AIF).**
  Let $M_d(C_i)$ be the number of methods declared in a class i
  Let $M_i(C_i)$ be the number of methods inherited (and not overridden) in a class i
  Let $M_a(C_i) = M_d(C_i) + M_i(C_i)$ = Number of methods that can be invoked in association with class i

$$\mathrm{MIF} = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Inheritance Factor – MIF and AIF cont..

Let $A_d(C_i)$ = Number of attributes declared in a class i
Let $A_i(C_i)$ = Number of attributes from base classes that are accessible in a class i
Let $A_a(C_i) = A_d(C_i) + A_i(C_i)$ = Number of attributes that can be accessed in association with class i

$$\mathrm{AIF} = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Example 2 – Calculate MIF and AIF from the C++ code below

```
Class A{
protected:
   int a;
public:
   void x();
   virtual void y();
};
Class B public A {
   int b;
protected:
   int bb;
public:
   void z():
   void y();
   void w();
};
Class C public B {
   int c;
   void v();
};
```

class	Md	Mi	Ad	Ai
A	x(),y()	none	a	none
B	w(),z(),y()	A::x()	b,bb	A::a
C	v()	B::w(),z(),y	c	B::bb
		A::x()		

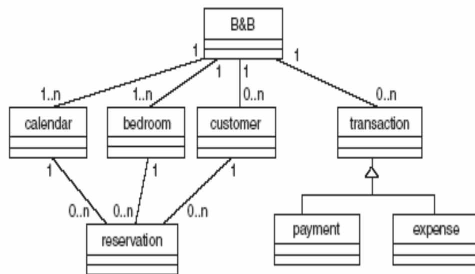
MIF = 5/11 AIF = 2/6

Coupling Factor (CF)

- **The coupling factor (CF) measures the coupling between classes excluding coupling due to inheritance.**
- **Let *is_client(C_i, C_j) = 1* if class *i* has a relation with class *j*; otherwise, it is zero.**
- **The relationship might be that class *i* calls a method in class *j* or has a reference to class *j* or to an attribute in class *j*.**
- **This relationship cannot be inheritance.**

$$\mathrm{CF} = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} is_client(c_i, c_j)}{TC^2 - TC}$$

Example 3 – Calculate the coupling factor on the object model shown below for B & B problem. Only assume a relationship if it is required by the association shown on the diagram



TC = 7

Class	is_client classes
B&B	calendar, bedroom, customer, transaction
calendar	reservation
bedroom	reservation
customer	reservation
transaction	none
payment	none
expense	none

CF = 7/42

Polymorphism Factor (PF)

- **The polymorphism factor (PF) is a measure of the potential for polymorphism.**
- **Let $M_o(C_i)$ be the number of overriding methods in class *i*.**
- **Let $M_n(C_i)$ be the number of new methods in class *i*.**
- **Let $DC(C_i)$ be the number of descendants of class *i*.**

$$\mathrm{PF} = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}[M_n(C_i) \times DC(C_i)]}$$

Example 4 – Calculate the polymorphism factor on the C++ code from example 2 above:

Class	Mn	Mo	DC
A	x(),y()	none	2
B	w(),z()	y()	1
C	v()	none	0

$$\mathrm{PF} = 1/(2*2+2*1+1*0) = 1/6$$