FUSION NEWSLETTER

Hewlett-Packard | Volume 5.3 | September 1997

FUSION UPDATE PART II: ARCHITECTURE AND DESIGN IN THE NEXT GENERATION OF FUSION

by Derek Coleman, Todd Cotton, Chris Dollin, Paul Jeremaes, Matt Johnson, Ruth Malan, David Redmond-Pyle and Herman Yang

Introduction

This article is part II in a series that is being used to keep the Fusion community apprised of developments as the method team migrates Fusion to the UML notation and adds strong support for architectural design and team development.

To recap from the last issue: This next generation of Fusion is described in terms of the engineering process and a complementary team management process.

The management process relies on the Fusion engineering models as a basis for effective project leadership, planning and coordination, and incorporates industry best practice. A high-level summary of the management process was presented in the February issue of the Fusion Newsletter.[1]

The engineering process focuses on supporting software development activities in requirements analysis, architecture, and design. It is written as if performed in sequence, but on an actual project the activities are organized according to the evolutionary delivery principles explained in the management process.

Fusion has five separate activities: Requirements, Analysis, Architecture, Design and Implementation (see Figure 1). A summary of the requirements and analysis phases was presented in the May issue of the Newsletter. This article takes over where the last one left off, describing the architecture, design and implementation phases.

1. "Evolutionary Development Update", *Fusion Newsletter*, vol 5.1. Available on the web: http//www.hpl.hp.com/fusion

Architecture

An *architecture* is a specification of the system to be built in terms of components and their interconnections. The architecture phase produces architectural descriptions at two different levels of abstraction. A *conceptual architecture* describes the system in terms of component collaborations which are expressed informally and at a high level of abstraction. The primary delivery of the

IN THIS ISSUE

New Fusion
Fusion Update: Architecture, Design & Implementation . 1

Requirements
Using Viewpoints to Aid Requirements Elicitation.......... 8
Defining Nonfunctional Requirements 10

Classic Fusion
The Fusion Process from an OO Perspective 14
Two Years of Fusion... 18
Estimating with System Operations 21

Announcements
Fusion Interest Group Meetings..................................... 23
New Book 23
For Further Information 23

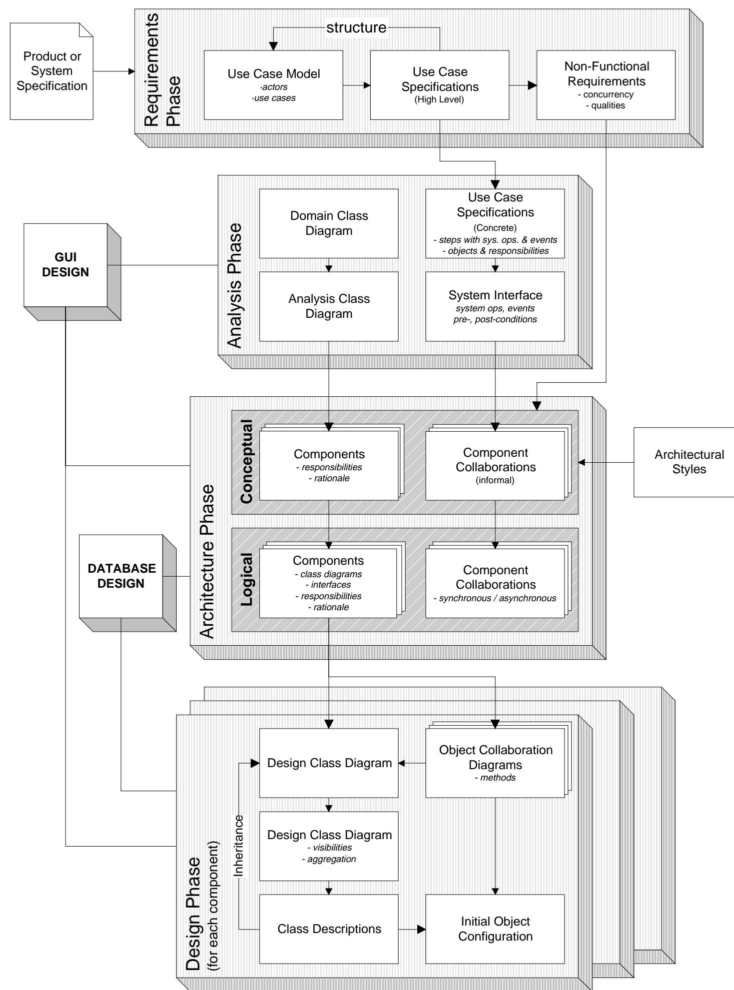

Figure 1 The new Fusion Engineering Process detailing the dependencies between models. (Note: the evolutionary process showing the sequencing of activities is shown in Figure 3.)

architecture phase is a *logical* architecture, which specifies collaborations in terms of messaging interfaces.

In the logical architecture, components are specified in the same way as a system, i.e. an analysis class diagram, and an interface consisting of a set of operation specifications and events. Consequently the *architectural phase can be applied recursively* to produce an architectural design at any desired level of granularity.

Step 1—Review and select applicable architectural styles.

An architectural style defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines a vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. Standard styles include layers, pipe-and-filter, blackboard and microkernel. The style of the architecture of a specific system is likely to be a hybrid of standard styles.

Review applicable architectural styles for the domain of concern and create a coherent architectural style by composing and refining pre-existing styles. The architectural style will constrain the decisions made in the subsequent steps of the process. A key criterion for evaluating architectural styles is satisfaction of the non-functional and scale requirements identified in the Requirements phase.

Step 2—Informal design of the architecture.

The goal of this phase is to make a first cut at the architecture. The description should be informal and provide a basis for the more detailed explorations of the later phases of the architectural process.

The chosen architectural style provides the framework

for the architecture. At this stage, the architecture may be described by a

- List of the components, their responsibilities and the rationale for the choice. Enter the information into the data dictionary.
- Description of the behavior of the architecture. Use diagrams to show the behavior in terms of dataflows or events passing between components. Often, box and arrow diagrams are better than using a formal notation.

The description should focus on how the architecture meets the requirements associated with areas of technical difficulty or high risk. It should be at the level of architectural invariants and mechanisms rather than on detailed component collaborations. If no architectural style was chosen in Step 1, then revisit the step to make the architectural style explicit.

Guidelines for identifying components include:

- Candidate components may be invented by subsetting the analysis class diagram. Classes are grouped together so that components are internally cohesive and loosely coupled externally with respect to associations and likely use. An instantiation of the classes will produce a component.
- Specialized components may be needed to support non-functional requirements, e.g. an authentication server to provide security.
- Review existing reusable libraries, and existing systems, to see if useful components already exist. Such *reusable* or *legacy components* can be used in the design process with their interfaces 'as-is' or modified, (e.g. using a wrapper).
- If use cases require the system to interact with human users, then a user interface component will be needed and should be designed according to the GUI design process
- Candidate components may also be introduced to support clusters of related responsibilities or system operations, for example those associated with a feature.
- If data needs to be shared between many use cases that are not temporally co-incident, then it must be determined if the data must be saved across invocations of the system using a database component or just within a single run of the system using a data structure. The database component should be designed using the Database design process.

Step 3—Develop the Conceptual Architecture.

The conceptual architecture describes the architecture much more rigorously and precisely. The increased precision should be directed at areas of highest risk or difficulty identified in Step 2.

This step details how the components interact in order to satisfy the requirements. Each scenario of use is used to construct a component collaboration diagram, which documents end-to-end behavior of the architecture. The collaborations between components, for example the delegation of responsibilities, are indicated by links between components. In a conceptual architecture, the links usually do not show message flows unless they are already known, as for example with legacy components.

Scenarios may be derived from use cases, or if a more precise understanding of architectural behavior is required from the pre- and postconditions for individual system operations. Design alternatives, for example to meet the non-functional requirements, can be explored and evaluated by developing different collaboration diagrams.

Enough scenarios should be explored to ensure that all components and collaborations have been discovered, and the architecture can support the requirements.

An *architecture diagram*, which provides a summary description of the system in terms of the components and their interactions, is constructed incrementally from the collaboration diagrams. In UML terms, an architecture diagram is an object collaboration diagram which provides the *context* for the set of collaborations between the components.

Developing the collaboration diagram for a scenario may refine the architecture by

- suggesting new components,
- adding new responsibilities to a component, e.g. designating a component as controller for a system operation, or
- introducing new collaborations between components.

Any new component, responsibility, or collaboration should be added to the architectural diagram and to the set of component specifications.

Components that are created and destroyed dynamically are often needed in architectures that have to support use cases with many concurrently active instances.

In a collaboration diagram, links may be annotated with

- *sequencing information*, i.e.the order in which the collaborations occur,
- *data flows* that occur as part of the collaboration
- *create* or *delete* which denote component creation and deletion
- *directionality* of the collaboration, i.e., which component initiates the collaboration

Guidelines:

- The architectural design process may reveal inconsistencies in style between use cases that can cause an unnecessarily baroque architecture. It is usually desirable to make the use cases less idiosyncratic in order to produce a cleaner architecture.

These use cases supersede analysis use cases and must be documented and rechecked with the customer.

Step 4—Develop the Logical Architecture

In this step the component collaborations of the conceptual architecture are replaced by message flows. Such architectural decisions cannot be finalized by looking at collaborations or components on an individual basis. It is necessary to take a more holistic view and consider how the decisions made for one collaboration can affect all the other collaborations involving the same components. Thus the conceptual architecture should be reviewed in order to establish relevant principles, mechanisms and/or patterns to guide the logical architecture design.

Using these principles, the logical architecture is developed by examining all the links on all the component collaboration diagrams developed for the conceptual architecture and deciding for each collaboration:

- whether the collaboration is mapped to a single message or a messaging protocol
- which component *initiates* the message
- whether the messaging is *synchronous*, *asynchronous* or can be left *unspecified* until the Design phase.

Prioritize the system operations according to real-time, throughput constraints and all the non-functional requirements. For each system operation, make the appropriate decisions for the collaborations involved. If necessary, develop the collaboration diagrams further to explore the behavior of the architecture. In order to evaluate how the architecture can meet time-critical system level requirements such as throughput, UML *sequence diagrams* may be used to explore the length and complexity of end-to-end interactions through the architecture.

The pre- and postconditions and the corresponding collaboration diagrams should be examined to consider whether it is possible for there to be simultaneous read/ write access to a component. If so, access to the component may need synchronization and this should be added to the component specification.

As each inter-component collaboration is refined into a method invocation it should be given a pre- and post-condition specification as in the Analysis phase. The objects referred to in the results and assumes clauses must appear in the class diagram for the component.

To summarize, a logical architecture is specified by:

1. an architecture diagram.
2. a revised set of collaboration diagrams showing message flows.
3. For each component:
    - an interface specification documenting the message interface. Each message is specified by a pre- and postcondition, as in the analysis phase.
    - an analysis class diagram
    - a list of responsibilities and the underlying rationale
    - whether it is dynamically created and deleted
    - whether it may require mutual exclusion

Guidelines:

- *Asynchronous vs. Synchronous communication.*
  Asynchronous communication between components should be introduced in order to allow time consuming computations to be performed “off-line” by another process or thread. This is particularly important when maximum responsiveness is required in interacting with actors. Introducing asynchronous communication can also affect interfaces because in order for a component to receive the results of an asynchronous computation there must be an event to transmit the results back. Further reasons for making interactions asynchronous include:
    - interaction with a resource component, i.e. a server that can receive requests from multiple clients
    - handling operations from the environment that are not constrained to appear in any particular order
    - when the client does not care when the required operation gets done
- *Interfaces*.
  Decide whether each component needs a *real* interface. A component with a real interface can enforce encapsulation by ensuring that its internal components are only accessed via that interface. Alternatively, a component can be treated as a white box that acts solely as an operational grouping of lower level components. In this case its interface is virtual, its internal structure is freely accessible to the environment and its interface is the union of all its internal component interfaces.
    - A component intended as a unit of implementation or distribution, should have a real interface.
    - If the interface of a component is essentially equivalent to the union of the interfaces of its components, then it may be preferable give the component a virtual interface.

  Decide whether two components can be merged. It may be desirable to merge two components if they are tightly bound to each other, have little functionality of their own and collaborate with very few other components.
- *Precision*
  The level of precision in describing component interfaces may vary depending on the nature of the project:
    - Leaving some component interfaces only partially

specified is appropriate for smaller-scale systems developed by small co-located teams. The complete details of the interfaces can be captured bottom-up, during the design of the components themselves. Of course, completing the design of the architecture in this case relies on good communication and disciplined attention to the component interfaces during design.

- Fuller and more precise interface specifications are more important when:
  - the components will be developed by teams that are geographically or organizationally distributed (e.g., the development of the component is to be out-sourced)
  - the components are units of distribution, plug-and-play interoperability and/or reuse
  - complex forms of concurrency need to be addressed, especially in dealing with real-time constraints

Step 5—Rationalize the architecture

Assess whether the proposed architecture can satisfy the quality requirements and any other non-functional requirements. Apply the measures and test scenarios that were developed in Step 6 of the Requirements phase, and ensure that the architecture meets these requirements. Identify architectural risks, e.g. points where performance is critical. These are candidates for early architectural prototyping.

Consider the architecture against these criteria:

- Does each component have clearly defined responsibilities? Are there components with a surprisingly large number of interactions? If so, they may need to be merged and re-analyzed. Are there risks of deadlock or races?
- Are the pre- and post condition specifications of each system operation satisfied by the architectural collaborations? If not, the architecture does not meet its functional requirements.
- Check whether the architecture can be mapped to the intended physical architecture. Allocate components to logical processors in the expected execution environment and check interaction path lengths for the scenarios used in constructing the architecture.

Step 6—Form Design Guidelines

Before entering the design phase it is necessary to establish any principles that must be adhered to by the designers. The guidelines provide for consistent design approaches across the system such as preferred communication mechanisms, security policies and error and exception handling. For example, a design guideline might require all system critical intercomponent messages to return a Boolean value indicating whether the method was successfully invoked.

Design

The outputs from the design phase are a design class diagram, object collaboration diagrams, and initial object configurations. During design, object-oriented structures are introduced to satisfy the abstract definitions produced from analysis and architecture. The design is in terms of design objects which are instances of design classes. Unlike an analysis class, a design class can have operations associated with it, and possibly additional attributes.

Step 1—Form the initial design class diagram.

The *design class diagram* shows the classes that are used during the design phase of a component. It is formed initially by making a copy of the analysis class diagram for the component.

Step 2—Construct object collaboration diagrams

The purpose of this step is to define object-oriented algorithms that satisfy the analysis specifications for the operations. Before designing the algorithms it is appropriate to review and evaluate whether there are any patterns that are applicable.

The algorithms are represented as *object collaboration diagrams*, one diagram is designed for each operation that the component is responsible for. The algorithm may involve *multi-threading* and the messaging model is *procedure call*.

The objects and associations mentioned in the *reads* and *changes* clauses of the operation specifications help determine the design objects involved in the algorithm. Often analysis objects can be mapped directly to objects of the corresponding design class. However, sometimes they may be mapped to one or more objects from new design classes, making the copied analysis class redundant. Each analysis association may be mapped to an object of a new design class that represents the association, or the association may be represented by attributes of objects involved in the association. Any new classes or attributes are added to the design class diagram.

The object that initially responds to the system operation is called the *controller*, and the others are the *collaborators*. The system operation is added to the interface of controller class on the design class diagram. The algorithm determines the messages that flow between objects and the data carried by the messages. When a method is added to a controller object it should also be added to the interface of the corresponding design class.

An object collaboration diagram may be drawn for each method on a collaborator class by treating the method

as a system operation. Consequently the design process can be used recursively to layer the design. The object collaboration step is complete when all methods on objects are sufficiently trivial that they may be safely left to the implementation stage.

The object collaboration diagram for a system operation, or a method, is the design artifact corresponding to the code of the method in some class. Consequently the diagrams must be mapped, or cross-referenced, to the class inheritance structure, in order to provide the design documentation for the virtual methods of superclasses and the inherited methods of subclasses.

Each *unspecified* method, that the component is responsible for, must be resolved into either a synchronous or asynchronous method. The object collaboration diagrams for the alternatives provide a basis for the decision. The decision has to be consistent with the architectural guidelines and the design of all the other components that use this method. Update the architecture to reflect the decisions.

The object collaboration diagrams should be examined to consider whether it is possible for there to be simultaneous read/write access to a object. If so, the object will need to ensure mutual exclusion on the appropriate methods, for example by using critical regions. If the component is specified as requiring mutual exclusion then each object that appears in more than one object collaboration diagrams should be examined to see if simultaneous access is possible. Simultaneous access can also occur as the result of two threads within a single collaboration diagram passing through the same object.

Guidelines:

- Ensure that any design classes imported from the analysis class diagram are still needed—they may have been made redundant.
- Consult with the architect and designers of the other components before deciding how to resolve an unspecified system operation.

Step 3—Object aggregation and visibility

All the object collaboration diagrams are inspected. Each message on an object collaboration diagram means that a visibility reference is needed from the client class to the server object.

Decide on the kind of visibility reference (bound/ unbound, permanent/transient, fixed/changeable, etc.) required, taking into account the lifetime of the reference, the lifetime of the visible object, and whether the reference can change once established. Use this information to decide whether aggregation inherited from the analysis phase should be mapped to aggregation by reference or value.

Record these decisions by adding the appropriate object valued, and object reference valued, attributes to classes on the design class diagram.

Guidelines:

- Check consistency with analysis models. For each association on the analysis class diagram, and that is used in a system operation specification, check that there is a path of visibility for the corresponding classes on the design class diagrams.
- Check mutual consistency. Ensure that exclusive target objects are not referenced by more than one class and that shared targets are referenced by more than one class.

Step 4—Rationalize design class diagram

Search for common objects, common classes, and common behaviors. Consider:

- if the objects belonging to a class have been given similar methods, can their methods be unified?
- if objects belonging to different classes have similar behavior, can the classes be unified, e.g. by introducing generalizations/specializations of each other or some new class?
- Do objects from a class have separable behaviors? If so, should the class
  - be split into separate classes,
  - be turned into an aggregate class,
  - or multiply inherit from several parent classes?

Would plausible changes in the requirements (analysis, architecture) affect several classes? Does this suggest changes to the class structure? Would plausible changes in the class structure make future maintenance easier?

If necessary revise the design class diagram, and object collaboration diagrams.

Step 5—Define initial object configurations

Use the object collaboration diagrams and the known life cycle of the system to decide which objects must be present when the component starts. The *initial objects*, and the *values of their attributes*, are the seeds from which the component will grow; they must be allocated either when execution starts, or when some agreed entry point of that component is entered.

Step 6—Review design

Verification of functional effect. Check that the functional effect of each object collaboration diagram satisfies the specification of its system operation given in the operation model.

Implementation

The final stage of Fusion is mapping the design into an effective implementation. Because most of the global decisions have been made during the architecture and design stages, this transition is relatively straightforward, although some of the issues it raises are subtle. Implementation is organized around components, but components whose design class diagrams share common classes may be able to share target code as well.

To begin coding, it is useful to generate a class description view of the design class diagram. The class description view is a summary of the information already present in the data dictionary and the other diagrams, and for each class defines:

- The operations from the interactions with their parameters and results
- The data attributes used by the operations of the class
- The parent class(es) of the class. Usually these will include supertypes from the analysis.
- The permanent visibility references of that class

Step 1—Resource management strategy

Consider object lifetimes and how the resources of objects that have become inaccessible are to be reclaimed. The most obvious such resource is storage; other resources include file handles and window descriptors. Decide how resource management is to be done within the component. If the target language has garbage collection, this makes store management much easier, but it may not be sufficient to cope with other system resources.

Resource management decisions are typically not local to a class, and depend on the target programming language. Thus they should be considered *before* embarking on the rest of the coding activity. Note also that resource consumption may have been one of the quality criteria for the delivered system.

Step 2—Code arising from data dictionary

Implement any types, predicates, and functions that are present in the data dictionary and used by classes in this component. In languages (such as Java) where functions cannot exist outside classes, allocate one or more design classes to hold those functions.

Identify assertions that will affect methods from multiple classes. Decide how these assertions should be reflected in the code.

Step 3—Code class descriptions

(Tool support may make this step automatic.) The class descriptions from the design phase provide the basis for the generation of classes in the component's programming language. Where the implementation language or its libraries provide an existing class suitable to the purpose, that class should be used (subject to performance or maintainability arguments).

Each attribute of the design class becomes a slot in the implementation class. The mutability and sharing properties of that attribute control target language annotations such as bold and private.

Each operation of the design class becomes a method of the target class.

Each parent class of a design class defines a parent target class of the target class.

Step 4—Code method bodies

Each method is implemented using the specification provided by its DCD or operation specification. Decisions must be taken about error handling and recovery; these decisions should be across entire components, not per-class. Iterations over collections must respect the DCD semantics. Wherever a method makes changes that might violate an assertion, consider how that change is to be implemented safely.

Step 5—Performance analysis

Performance is often one of the quality requirements of a system. Performance cannot be obtained as an afterthought; it must be considered throughout the analysis, design, and implementation process. However, until code is in place, it is difficult to anticipate its performance. Remember that *optimizing rarely-executed code is ineffective*.

So when performance is an issue, *profile your system* in as many ways as you can, and plan to make this easy when you start the implementation activity. Isolate the hot spots where performance leaks away, and optimize those.

Step 6—Code review

Code is reviewed to establish its correctness and quality, and to anticipate and avoid future maintenance problems. There are two styles of code review:

Inspections. A cost-effective technique for the detection of defects in software. Code is reviewed by teams which the intention of detecting (not correcting; that comes later) possible problems. Object-oriented code is more difficult to review than traditional code, because analysis of the flow of control is complicated by the typically small size of methods, their dispersal among classes, and the use of both static and dynamic polymorphism (which can make

it hard to bind a call to its implementation). Object-oriented inspections should also focus on detecting typical flaws in object-oriented systems, e.g., they should ensure that all subclasses implementing a specific method conform to the method specification.

Testing. A complementary technique to inspections for exposing defects in software. Test cases for Fusion can be driven by use cases.

- Applying algebraic properties such as associativity and identity preservation to member function invocation.
- Checking the destructors in C++, are consistent with the corresponding constructors.
- Ensuring that Java classes which claim non-memory resources are equipped with finalizers.
- Checking proper use of initialization.
- Checking that casting in C++ is being used in safe ways.
- Trying to trigger exception handling capabilities via extreme boundary value inputs.

Copyright 1997 Hewlett-Packard Company

ESTIMATING WITH SYSTEM OPERATIONS

by Kris Oosting,

Estimation: an art, a science, rules of thumb or the big something? Well, probably a little of all of them.

One of the beauties of Fusion when it was introduced in 1994 was its simplicity. This makes life easier when estimating the amount of work needed to create all the models.

We found that the *system operation* provides a nice piece of information for estimation allowing us to predict how long it would take to design and build the system.

We first used simple formulas to estimate, and later these formulas were tuned. Now that we have tuned the formulas for 'standard' Fusion and have gained experience, the method changes its process and notation. This is both good and not so good. It is good because we have to move on. Users' needs change, developers' needs change, and technology changes. It is less good, because we again have to collect a lot of metrics in order to understand how long developers are working to produce certain models, how defect prone the models (or process) are, etc. We can of course use the basic formulas, but have to tune them and change them again. The risk is that when we are able to estimate properly again, the method and/or notation changes once more.

Well, not to be dramatic about it, the bottom line is that we have to collect metrics to understand what we're doing.

As you know by now: “You can control what you can measure.”

Let’s look at the formulas and the items that are important.

System Operations

The number of system operations can be seen as the amount of functionality that has to be implemented by the system. The life-cycle model can be seen as the model that shows “dependencies” between the system operations (i.e. the functionality).

The number of system operations can be found in the requirements, graphical user interface, etc. Even quite early in the process an indication of what and how many system operations can be found.

The problem with some projects is that we want to know exactly how long it takes, how much it costs, before the project has even started or before the requirements are finished. This is different however when using the TeamFusion process.

System Operation Category

On many projects we used the following formula to estimate the time to design the system:

*# system operations * 2 hours = total design time*.

Very simple and very basic. This covers making object interaction graphs, visibility graphs, inheritance graphs including their descriptions.

For some time this was sufficient. Later we measured that some system operations took 30 minutes to design while others took 30 hours to design.

That’s why we introduced *system operation categories* (SOC). These SOCs group system operations depending on the type of their function. For example: system operations that have to do with fetching data, or to start complex calculations.

A system operation that initiates the collection of data from a radar, creates an image, and reports back is much more time intensive to design (and defect prone) than a system operation that simply fetches a selected customer from a database (using objectware of course).

What are these system operation categories you might ask. Well, very simple, re-use is the answer.

We started with the Process Categories from Tom DeMarco from his book Controlling Software Projects (1982!)

These are:

Category	Description	Weight
Separation	primitives that divide incoming data items	0.6
Amalgamation	primitives that combine incoming data items	0.6
Data Direction	primitives that steer data according to a control variable	0.3
Simple Update	primitives that update one or more items of stored data	0.5
Storage Management	primitives that analyze stored data, and act based on the state of that data	1.0
Edit	primitives that evaluate net input data at the man-machine boundary	0.8
Verification	primitives that check for and report on internal inconsistency	1.0
Text Manipulation	primitives that deal with text strings	1.0
Synchro -nization	primitives that decide when to act or prompt others to act	1.5
Output Generation	primitives that format net output data (other then tabular output)	1.0
Display	primitives that construct 2-dimensional outputs (graphs, etc.)	1.8
Tabular Analysis	primitives that do formatting and simple tabular reporting	1.0
Arithmetic	primitives that do simple mathematics	0.7
Initiation	primitives that establish starting values for stored data	1.0
Computation	primitives that do complex mathematics	2.0
Device Management	primitives that control devices adjacent to the computer boundary	2.5

So the new formula becomes:

$$TotalDesignTime = \sum_{i=0}^{n} (so_i \times cw) \times t$$

where:

n: number of system operations

so: system operation

cw: category weight

t: “typical” time to make the design for one system operation (5 hours)

What you must do is to find the right values of Weight (table above) for your organization.

Now we have the time to make the design, but what about the time to build from it. From experience we’ve created the following rules of thumb (using standard Fusion):

For C++/Motif projects:

if Design Time = 1 **then** Time to build is between 1.6 and 2.1 and analysis time is ± 0.8

For Objective-C/NEXTSTEP Projects:

if Design Time = 1 **then** Time to build is between 0.5 and 1.2 and analysis time is ± 0.8

For example when you calculated that doing design would take 115 hours, then using the first rule of thumb, building it would take between 184 and 241.5 hours. Analysis therefore would take around 92 hours.

As we get so many questions about the relationship with Function Points, we are now collecting metrics to make that connection. Another issue is the relationship with the number of defects we can expect. This means that very early in the process we can do risk calculations and can prepare the resources needed to remove the defects.

We have some good metrics on the subject, but find it a little too early to publish them. As soon these metrics are ready to use, they will be published in an article.

When using the formulas in this article, please remember to calibrate the weights and "typical" time to design one system operation. These can be different per company or application type (real-time vs. administrative) or even differ per development environment. We always saw very good results when using NEXTSTEP (OpenStep).

Success with your estimations!