A Beginner's Guide to Developing with the Taligent Application Frameworks

Joachim Laubsch

September 6, 1995

Application Engineering Department
Software Technology Laboratory
Hewlett-Packard Laboratories
1501 Page Mill Road, Bldg. 1U-14
P.O. Box 10490
Palo Alto, Calif. 94303-0969

laubsch@hpl.hp.com
(415) 857-7695

Abstract

The conclusions of a three months learnability and usability study of Taligent Application frameworks are presented in terms of a preparatory guide for a novice user. It has been known that learnability of large object-oriented systems poses a serious obstacle to adoption of object-oriented technology. Taligent frameworks face this obstacle in particular. This paper was written to help you plan your learning path.

Keywords

Application development, application frameworks, learning of frameworks, Object-oriented programming and design, Taligent frameworks.

Contents

1 Introduction

During a period of three months I learned to use Taligent application frameworks, in particular the Presentation Framework and implemented a simple application. This report summarizes some of the lessons learned, and points out sources of difficulties. The next section explains why learnability is an important issue affecting the market acceptance of frameworks. Section 3 gives a definition of frameworks and briefly characterizes Taligent's CommonPoint frameworks. Section 4 suggests a roadmap for your learning path and section 5 gives guidelines for developing a first application most effectively.

This report does not expose any of the frameworks, idioms, architecture diagrams etc if these are covered by the CommonPoint 1.0 documentation (see [Tal95b],[Tal95e], [Tal95a], [Tal95d], [Tal95g]). It is written not to require previous exposure to Taligent's documentation, but includes references to the relevant sources, as well as some references to foundational literature on frameworks, object-oriented programming, and OOAD methods.

2 Framework Learnability

Why is learning a problem? Reporting from his experience of teaching SmallTalk to engineers, Kent Beck stated in 1986:

> "Becoming familiar with the ~250 classes in the SmallTalk image can be a daunting task to a newcomer." [O'S86] p. 503

How will an application developer confronted with a system, larger by an order of magnitude cope with the learning of its reusable parts? The learnability issue will be a major factor determining the market acceptance of CommonPoint since in comparison to competing products, its area of coverage is much larger. This applies as well to objects of foundational services as well as at the application level. The following table gives an indication of this by simply counting the number of classes:

Product	*Number of Classes*
CommonPoint	1940
VisualWorks	745
OpenDoc	315
NextStep/OpenStep	128
MFC (MicroSoft)	131

Aside from size, complexity is increased due to the need for understanding interacting classes. In a survey we conducted, learnability was mentioned as a main inhibitor to framework use by developers familiar with frameworks (see [Lau95]), and early developers with Taligent experienced "a stiff learning curve" even for experienced C++ programmers [San95].

3 Frameworks and their Benefit

What is a framework?

> "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems." R.E. Johnson [Joh88]

> "A framework defines a subsystem or mechanism that is customizable or extensible. The subsystem *design* is encapsulated by a set of classes, though the *implementation* may only be partially specified. Framework abstract base classes leave some member function definitions to the application." [Cop91] p. 373

> "The term *application framework* is used if this set of abstract and concrete classes comprises a generic software system for an application domain." [Pre95] p. 54

A framework is a set of prefabricated components of a working program which already provides a minimum default behavior. Developers can use, extend, or customize it for specific solutions. They only write the code that extends or specializes the framework behavior to suit the application's requirements. A developer will build an application by first identifying the framework and then filling in some of the customizable functionality. The main data and control flow infra-structure is already predefined in the framework. In C++ frameworks, base class member functions take care of the sequencing of application-dependent functions which are implemented as private virtuals in the derived class.

But do frameworks have practical value? Application development is often not directed towards an individual application, but a family of applications with common base functionality. Application frameworks capture this genericity and provide for a disciplined way of specialization. Figure 1 gives an example. The framework includes a main event loop which calls the HandleUpdate function of TApplication which in turn calls DrawSelf on every TWindow object. The application gets this behavior simply by instantiating TApplication, and subclassing TWindow with its own DrawSelf method.

This is called framework *adaption* and elaborated in step 5 of the plan described in section 5.1 below.

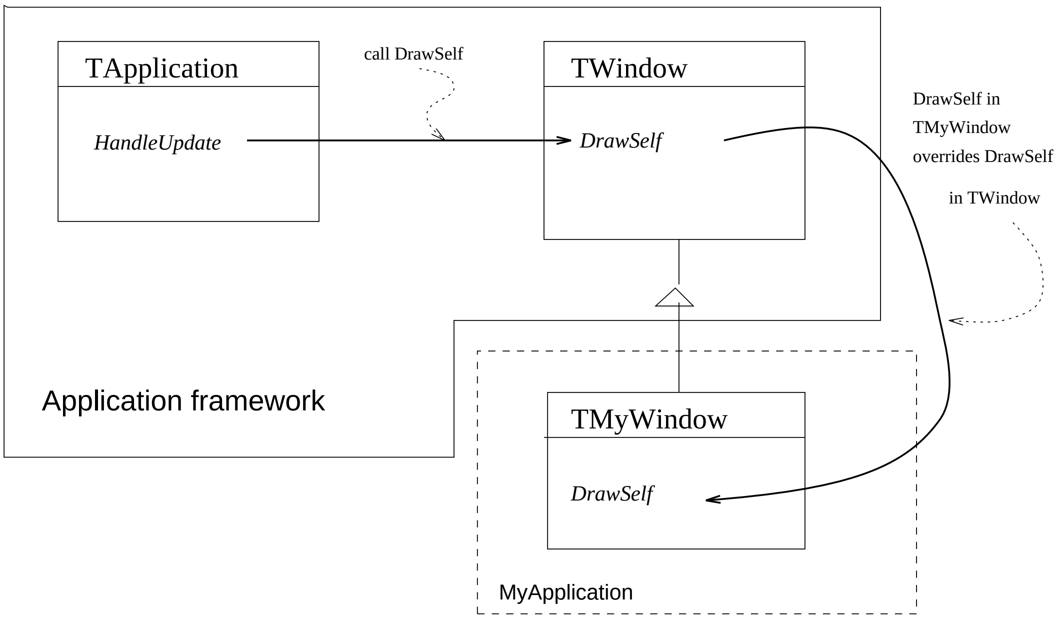

Figure 1: A simple application framework

As a consequence, a large amount of design and code is reused, making the transition to object-oriented technology more effective than could be achieved by a mere switch to an object-oriented language.

Taligent's CommonPoint frameworks CommonPoint 1.0 includes about 100 frameworks divided into two groups:

1. Application frameworks provide features for developing interactive applications.
2. System Services comprise a set of services, on which Application frameworks are built.

Application frameworks contain at the highest level a set of "Desktop Frameworks", with the *Presentation Framework* being one of them. Since the *Presentation Framework* unifies a number of document model and user interface mechanisms, it is likely to be **the** generic application to be specialized by an application developer.

Aside from frameworks covering entire application families, there are frameworks for particular problem areas, falling into the area of System Services. CommonPoint 1.0 examples of such frameworks are:

1. the Data Access framework which allows data on remote or local (SQL) databases to be accessed, queried or modified

2. the Notification framework which provides a mechanism to propagate change information from one object to another, or

3. the Caucus framework which provides for multicast communication facilities for collaborative applications.

Our experience with a small fraction of these frameworks showed that they are designed in a principled way, and implemented reliably. Taligent designers and engineers deserve praise on this point since:

> "Good frameworks are usually the result of many design iterations and a lot of hard work." [WBJ90]

4 A Roadmap to Learning Taligent Frameworks

Figure 2 shows a recommended roadmap to learning Taligent frameworks. The dotted arrows show optional material.

1. Prerequisites: A good mastery of C++, on the level of [Lip91] and [Cop91] is desirable. Eventually, you will have to understand functors (objects that behave like functions).

   You should have read "Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++" ([Tal94] available in bookstores). It teaches the fundamentals of OOAD, and introduces C++ programming conventions observed by Taligent. Previous OOAD background literature (such as [CAB$^+$94]) helps in understanding this design guide[1].

2. Before you start reading about Taligent's concept of frameworks in "Introduction to the CommonPoint Application System" (see [Tal95e]), or the recently published "Inside Taligent Technology" (see [CP95]), it would be useful to have some understanding of *design patterns* (see [GHJV94] or [Pre95]) since this trains you to think in terms of "purposeful structures of objects", as you will encounter them later in the exposition of application frameworks.

3. The CommonPoint printed documentation includes a self-guided tour manual: "Programming with the Presentation Framework: Tutorial" [Tal95f]. In particular, familiarize yourself with the architecture diagram notation explained there. It is easier to follow this tutorial if you had some previous exposition to an application framework such as MacApp, ET++, or InterViews. If not,

[1] An annotated bibliography can be found on `http://www.taligent.com/resources-list.html`

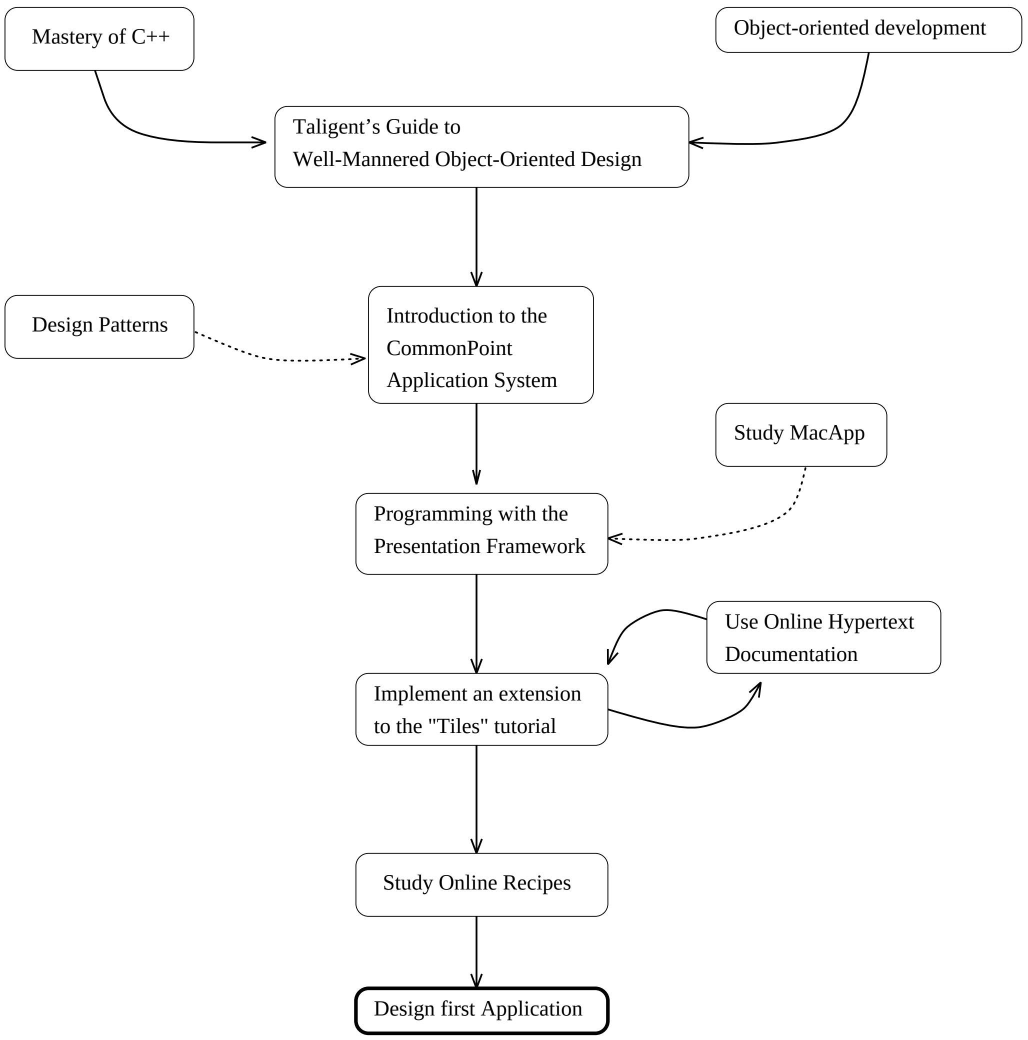

Figure 2: Roadmap to Learning Taligent Framework Use

study the MacApp framework (e.g. from [WRS90]), since many graphical user interface terms known from MacApp reoccur in the Presentation Framework.[2]

Read the tutorial trying to recognize Taligent's programming idioms you have learned in step 1 above (for examples see section 5.3 below). Taligent uses not only naming conventions but also C++ idioms you may not have come across. If you don't use these idioms, your code may work, but will be hard to follow by others, and break down when unforeseen functionality has to be added.

Implement some extensions to the application, for example change the model (e.g. add a size attribute to the tile object), or the extensions mentioned at the end of the tutorial.

During this tutorial it is very important to become familiar with Taligent's class diagram notation (in particular if you have not previously been exposed to it, say from [GHJV94]. It is helpful to practice drawing such diagrams for idioms encountered (if applicable) or the selected extensions.

The time needed for this is at least one month, and the experience of people who tried to shortcut this period has been negative.

Avoid starting to study the "Documented Samples" ([Tal95c] before having completed your first month of intensive training. (Reason: the samples are written by experts for demonstration purpose, without pedagogic intent.)

4. Use the online documentation to study available *recipes*. Figure 3 shows an example recipe. A recipe uses a concrete code snippet to show how an isolated problem can be solved. The recipes are organized in a hypertext model as a *cookbook*.

   After a short familiarization with the cookbook (2 to 3 days) you may sporadically return to particular recipes that your application development demands.

5. Learn to use the online "Class and Member Function" hypertext documentation. It will be your online dictionary for quickly determining the signature of a method, the protocol of a mixin class etc.. You will not learn about the functionality of any particular framework from the online "Class and Member Function" documentation. Consult the Developer's Guides for that [Tal95e].

5 Lessons Learned and Pitfalls to Avoid

This section helps you design and implement your first application with CommonPoint frameworks. In the following we emphasize the importance of architecture diagrams,

[2] A "cultural" heritage between Apple and Taligent is prevalent in many of the user interface concepts.

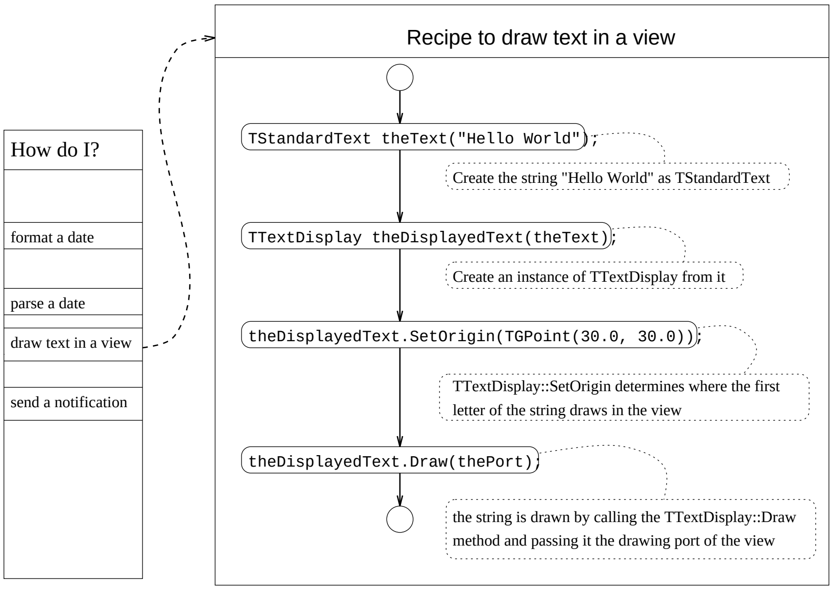

Figure 3: Hypertext cookbook of recipes

explain steps prior to the implementation of your application (after a requirement analysis and some object-oriented analysis), and finally mention some of the most important idioms to be followed during implementation.

5.1 Use architecture diagrams: Class and Object diagrams

> "We must spend more time in the future discovering, teaching, and writing about the general principles of object-oriented design". Dan Halbert in [O'S86]

Taligent frameworks expose design at a higher level of abstraction than the code. They model good design principles, and thereby have a teaching effect. Various types of diagrams are instrumental for exposing design: class diagrams, object diagrams and interaction diagrams.

CommonPoint framework documentation uses only class diagrams[3]. These document the static relations among classes. Interaction diagrams are used by CommonPoint only to document the sample applications [Tal95c].

CommonPoint documentation lacks *object diagrams* which represent the dynamic relations among objects[4]. Gamma introduced a flavor of object diagram notation in his

[3] originally introduced by Wilson [Wil90] and also explained in [GHJV94].

[4] In Fusion [CAB+94] the term "object interaction graph" is used.

thesis [Gam90] and some variant of it may become part of a future TalDE (Taligent Development Environment).[5]

Since we found object diagram notation very useful, and it is not included in the CommonPoint documentation, we briefly explain it here (if you know some other object diagram notation you can skip the rest of this section): an object diagram represents a snapshot of a network of objects who have references to each other, i.e. *know* each other. The *owns* relation can be indicated as a special case of the *knows* relation. Since C++ program design involves making decision about memory management, it is important to have an explicit account of the *owns* as well as *knows* relation.

Figure 4 is an example object diagram (from [Gam90]).

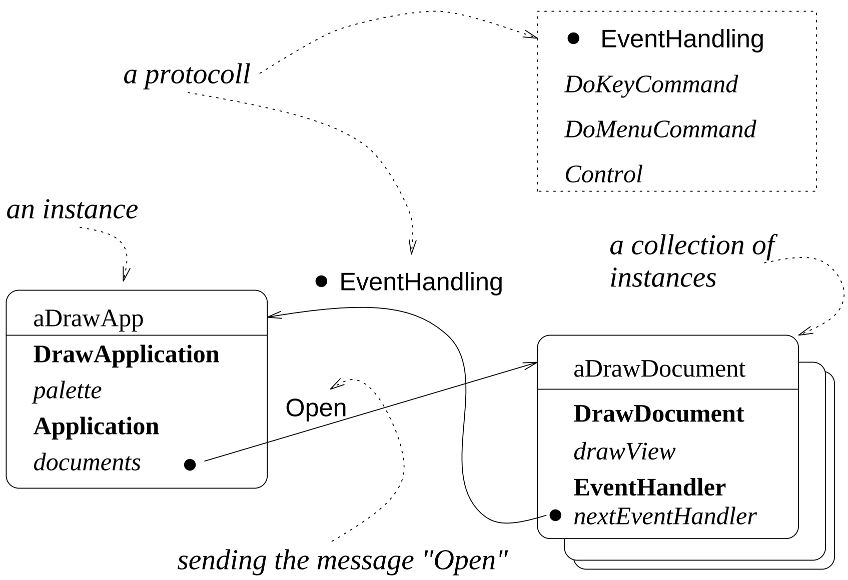

Figure 4: Example Object Diagram

Objects are depicted as rounded rectangles. The object name is separated from the rest by a line. Methods and fields are presented as in class diagrams, except that the hierarchy may be flattened, so that one sees the class a particular method is inherited from. An unlabeled arrow indicates reference (solid if it also implies ownership). A labeled arrow denotes message passing. A label preceded by a dot denotes a protocol. The messages of a protocol are shown in a dashed box — also called a *mixin class* in

[5]Unfortunately, the current CommonPoint documentation does not use object diagrams. A possible reason for Taligent's hesitation is that exposing too much control structure of the framework makes improvements more difficult. The CommonPoint 1.0 development environment will include LOOK, a tool for visualizing C++ applications. LOOK produces object diagram snapshots. I thank Erich Gamma for pointing out the benefit of the object interaction view for program understanding.

Taligent's terminology[6].

Object diagram notation is useful for program understanding and documentation, because it makes object sharing explicit, and helps you visualize a state of computation.

5.2 Make an Implementation Plan

Adapting one or more frameworks is the essence of implementation. This section describes steps for a process of framework adaption.

1. Initially, reduce the problem to a kernel and describe its functionality. Describe the layout of views, GUI elements, possible user commands, and their effect on the model and views.[7]

   Do not overspecify your design, since you may miss opportunistic use of framework facilities. For example, if your design calls for the user selecting a single element from a small set, you may decide which GUI element to use after having explored examples in Taligent's "Sample Applications".

2. Understand at a functional level the TalAE frameworks, that might be relevant. This means: understand the class diagrams of the frameworks relevant to your design, and know what their customizable functionality is, i.e. you must be able to point out their *hot spots*[8].

3. Choose the relevant framework(s) and express the design in terms of selected frameworks using CommonPoint's architecture diagram notation (see section 5.1 above). For example, if you chose the presentation framework the class TModel is one of its hot spots, and an application-specific model needs to be defined (see step 5 below).

4. Design for extensibility: Often there may actually be a set of framework classes that must be selected. If there are classes with similar functionality it is advisable to chose those with larger functionality, in case the kernel application is to be expanded later. For example, if you chose a class supporting a view

[6] A mixin class is a class you can use to add more functionality to a class that derives from another primary base class. Base classes represent fundamental objects (types) and mixin classes represent optional functionality (protocol).

[7] A good way to do this is to use the patterns language for the Model-View-Controller framework described in [GHJV94].

[8] For the technical meaning of this in terms of *template* and *hook* functions see Pree [Pre95]. The basic idea is that framework adaption occurs at points of predefined refinement, called *hot spots*. A template function is a method which supplies the flow of control and receives parameters for customization. (The term "template" has no relation to the C++ use of the word.) The template method calls *hook* methods, which need to be defined by the framework user, and capture the customization of the current application.

which can access the GUIBundle[9] over a class that does not, this view will have access to the model, presenter and selection, which may be important for some methods to be defined on this view.

5. **Adapting a framework**: Figure 5 shows the two APIs of a framework that an application program may interface to: the *calling API* and the *subclassing API*.

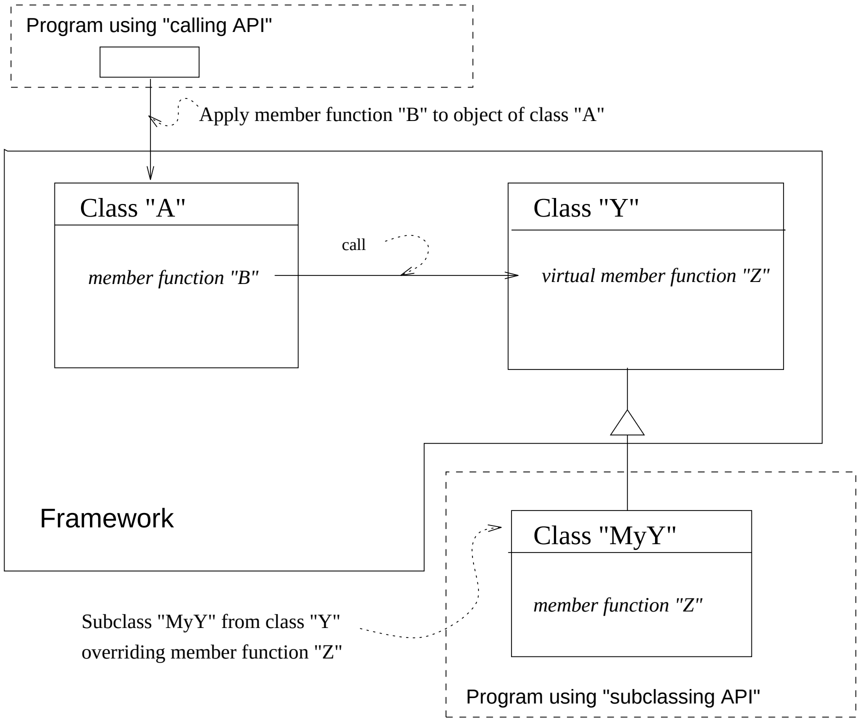

Figure 5: Calling API and Subclassing API

The calling API applies some method B (a *template function* in Pree's terminology [Pre94]) of the framework to an instance of framework class A. B in turn polymorphically calls Z on an instance of a subclass MyY of the framework class Y. Z (a *hook function* in Pree's terminology) has been defined in the application (possibly overriding default behavior of the framework).

A process for framework adaption follows: Start by selecting a class from the framework. Now there are the following alternatives:

(a) if this class contains all the needed functionality, directly instantiate from it (calling API),

[9] Think of a GUIBundle as an object via which the main participants of a Model-View-Controller architecture communicate.

(b) derive from the framework class if the class does not provide all the functionality needed or if the class requires to override some member (subclassing API),

(c) mixin from a framework. Understand which mixin classes exist for a framework (classes whose name starts with the letter 'M'). Objects collaborate by communicating with each other via a shared protocol, and TalAE defines individual protocols via *mixin* classes. Mixin classes can come from the same framework or different frameworks.

Any framework adaption via the subclassing API requires the developer to observe certain constraints:

- Pure virtual functions must always be implemented in the derived classes.
- virtual functions with a *default* behavior might not need to be implemented in the derived classes.
- virtual functions without *default* behavior must be implemented in the derived classes.

6. Decide object ownership. LaLonde and Pugh write ([LP95]):

   > "The three biggest problems for C++ programmers are forgetting to delete an object (i.e. losing space), deleting an object too soon (i.e. it is still needed), and attempting to delete an object more than once (you can't delete what you no longer have)."

   Augment the initial architecture diagram which abstracts from ownership with the ownership relation. This is necessary in a language like C++ which does not provide automatic garbage collection.

7. To clarify complicated cases draw "interaction diagrams" to capture flow of control. For instance, if you use container objects, the sequence of initialization is constrained by the fact that some initializations on the containee cannot be performed until it is adopted by the container. Such dependencies can be made explicit in interaction diagrams. For examples see [Tal95c].

5.3 Use Taligent Idioms

An idiom is a coding level pattern or programming cliche. If you understand Taligent idioms you will be able to grasp larger chunks of meaning at a time, and if you use idioms you and others will understand the code better. Taligent idioms also enable the reusability of components, e.g. if you observe the type extension idiom (see 3 below), your components can be part of objects which are persistently stored.

1. Never derive a base class just to add protocol (member functions). Define a mixin class instead. In CommonPoint data types are base classes and protocols are mixin classes.

2. Follow the *Law of Demeter* ([LHR88]) which has the purpose to reduce the amount of object coupling: In each method M of a class C only call methods of the following classes:

   (a) classes of C's members

   (b) classes of the arguments of M

3. Use the TaligentTypeExtension facility to declare that instances of that class can be saved as persistent objects. This gives these objects runtime type identification (RTTI). To accomplish this, aside from calling the appropriate declaration and definition macros, the following member functions need to be defined:

   (a) default constructor

   (b) copy constructor

   (c) assignment operator

   (d) stream in and out operators

   Observing the rules for defining these methods gives the benefits of dynamic type checking with efficiency and safety of static type checking! Coding is simplified by using the application template for the model definition of the presentation framework. They contain raw C++ source code which with a few find-and-replace commands can be adapted to your class declaration and definition.

4. Observe Taligent terminology for making object ownership explicit. Use the prefixes Adopt, Orphan, Create and Get consistent with Taligent's semantics! This helps avoid the three most common C++ errors mentioned above ([LP95]),

5. Use the iterator abstraction for collections:

   ```
   <TDeleterFor<TIteratorOver<Type>> iterator
   ```

   This declares the iterator and will delete it after it goes out of scope.

6 Conclusion

The time it takes to become a productive developer with Taligent frameworks is long (at least three months until you can approach your first application). Following the suggestions of this report will help you to assess the learning problem more realistically and make your learning experience more effective and enjoyable.

Since Taligent frameworks are a principled way to design reusable object-oriented components, you will not only be able to use such components, but also be well prepared to extend frameworks or design new frameworks to be reused by others.

References

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: the Fusion Method.* Prentice Hall, 1994.

[Cop91] J. Coplien. *Advanced C++: Programming Styles and Idioms.* Addison-Wesley, 1991.

[CP95] Sean Cotter and Mike Potel. *Inside Taligent Technology.* Addison-Wesley, 1995.

[Gam90] E. Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design.* Springer-Verlag, Berlin, 1990.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Joh88] R. Johnson. Designing reusable classes. *The Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Lau95] Joachim Laubsch. Taligent application frameworks: a technology assessment. Report, Hewlett-Packard Laboratories, June 1995.

[LHR88] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. *ACM SIGPLAN Notices – OOPSLA'88*, 23(11):323–334, 1988.

[Lip91] S.B. Lippman. *C++ Primer.* Addison-Wesley, 1991.

[LP95] Wilf LaLonde and John Pugh. Complexity in C++: A Smalltalk perspective. *Journal of Object-Oriented Programming*, pages 49–56, March-April 1995.

[O'S86] T. O'Shea. Panel: Learnability of object-oriented programming systems. *ACM SIGPLAN Notices (Proceedings of OOPSLA'86)*, 21(11):502–504, 1986.

[Pre94] Wolfgang Pree. Meta-patterns: A means for describing the essentials of reusable o-o design. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of ECOOP'94*, pages 150–162. Springer Verlag, 1994.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[San95] Rich Santalesa. Taligent readies a new development paradigm. *IEEE Software*, 12(2):103–105, March 1995.

[Tal94] Taligent. *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*. Addison-Wesley, 1994.

[Tal95a] Taligent Inc, Cupertino. *Desktop Frameworks concepts*, 1995.

[Tal95b] Taligent Inc, Cupertino. *Distributed Computing*, 1995.

[Tal95c] Taligent Inc, Cupertino. *Documented Samples*, 1995.

[Tal95d] Taligent Inc, Cupertino. *Foundation Services*, 1995.

[Tal95e] Taligent Inc, Cupertino. *Introduction to the CommonPoint Application System*, 1995.

[Tal95f] Taligent Inc, Cupertino. *Programming with the Presentation Framework: Tutorial*, 1995.

[Tal95g] Taligent Inc, Cupertino. *Text, Native Language Support, and Time Media*, 1995.

[WBJ90] R. J. Wirfs-Brock and R. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[Wil90] D. A. Wilson. Class diagrams: A tool for design, documentation and testing. *The Journal of Object-Oriented Programming*, 3(1):38–44, 1990.

[WRS90] David A. Wilson, Larry Rosenstein, and Dan Shafer. *C++ Programming with MacApp*. Addison Wesley, 1990.