**2**

**Initializing Applications**

This topic addresses what needs to happen to get a Crystal for WPF application up and running. There are two options for initializing WPF applications. One is extending Application class defined in System.Windows. Another is creating a dedicated bootstrapper. Both are to add functionalities to support modular composition, dependency injection, and other services.

A Crystal application requires registration and configuration during the application startup process—this is known as bootstrapping the application. The Crystal bootstrapping process includes creating and configuring a module catalog, creating a dependency injection container, configuring default region adapter for UI composition, creating and initializing the shell view, and initializing modules.

# Application and Window

WPF is built on two integrated components – User32 and DirectX. The **Application** class encapsulates application-specific functionalities, including Application Lifetime, Resource Management, and Navigation.

**Application** implements the singleton pattern to provide shared access to its window, property, and resource scope services. Consequently, only one instance of the **Application** class can be created per **AppDomain**.

# Core functionalities to Initialize

ds

# CrystalApplication – Extending Application

# Hello Crystal – Minimal Composite Desktop Application

This section builds a minimal composite desktop application via CrystalApplication, and shows the difference from default WPF application.

**Step 1: Create a WPF application with the default Visual Studio template.**

**Step 2: Add Assemblies Crystal.dll and Crystal.Unity.dll**

**Step 3: Change default App to derived class from Application, and remove StartupUri**

# CrystalBootstrapper – Dedicated Class

A bootstrapper is a class that is responsible for the initialization of an application built using the Prism Library. By using a bootstrapper, you have more control of how the Crystal Library components are wired up to your application.

The Crystal Library includes a default abstract **Bootstrapper** base class that can be specialized for use with any container. Many of the methods on the bootstrapper classes are virtual methods. You can override these methods as appropriate in your own custom bootstrapper implementation.



Figure 2.1 Basic stages of the bootstrapper process

The Crystal Library provides some additional base classes, derived from **Bootstrapper**, that have default implementations that are appropriate for most applications. The only stages left for your application bootstrapper to implement are creating and initializing the shell.

# Dependency Injection

Typically, developers of client applications face quite a few challenges. Application requirements can change over time. New business opportunities and challenges may present themselves, new technologies may become available, or even ongoing customer feedback during the development cycle may significantly affect the requirements of the application. Therefore, it is important to build the application so that it is flexible and can be easily modified or extended over time. Designing for this type of flexibility can be hard to accomplish. It requires an architecture that allows individual parts of the application to be independently developed and tested and that can be modified or updated later, in isolation, without affecting the rest of the application.

Most enterprise applications are sufficiently complex that they require more than one developer, maybe even a large team of developers that includes user interface (UI) designers and localizers in addition to developers. It can be a significant challenge to decide how to design the application so that multiple developers or subteams can work effectively on different pieces of the application independently, yet ensuring that the pieces come together seamlessly when integrated into the application.

Designing and building applications in a monolithic style can lead to an application that is very difficult and inefficient to maintain. In this case, "monolithic" refers to an application in which the components are very tightly coupled and there is no clear separation between them. Typically, applications designed and built this way suffer from problems that make the developer's life hard. It is difficult to add new features to the system or replace existing features, it is difficult to resolve bugs without breaking other portions of the system, and