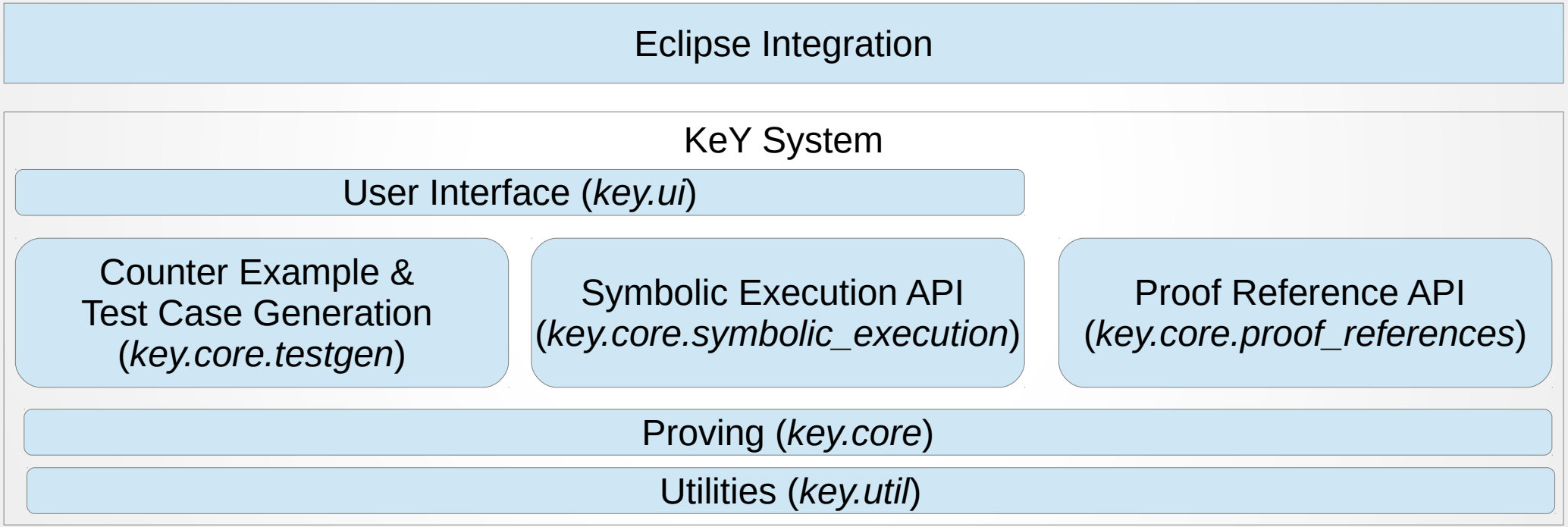
# KeY tool set

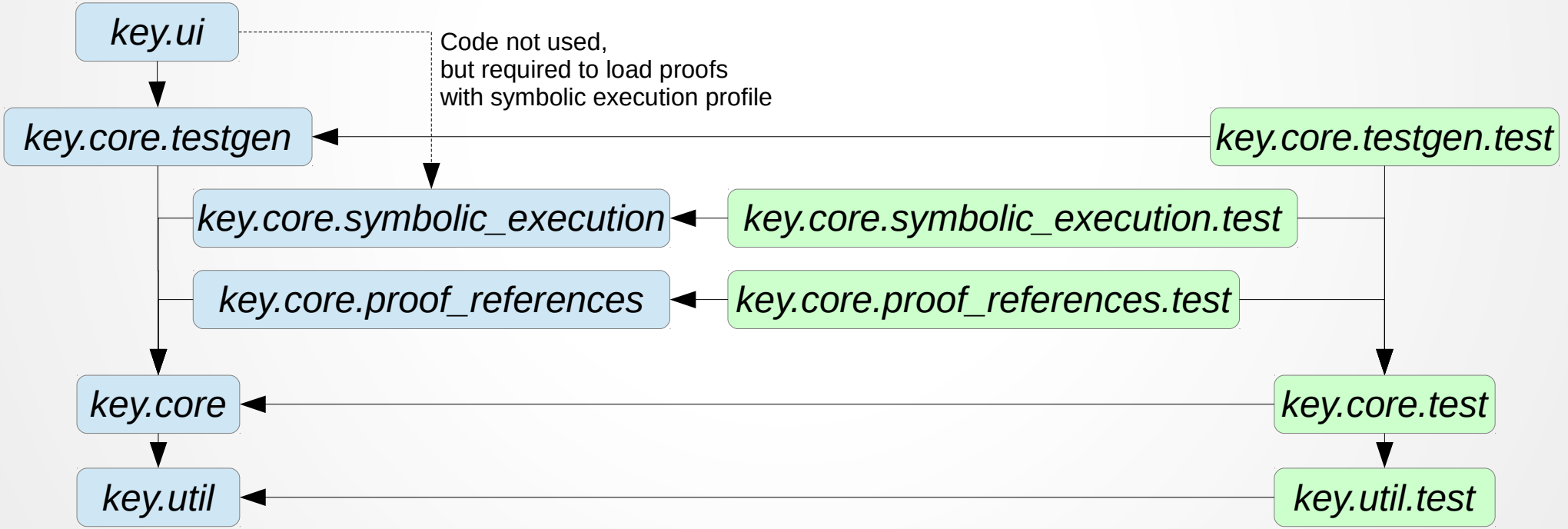

Eclipse Integration
KeY System
User Interface (key.ui)
Counter Example & Test Case Generation (key.core.testgen)
Symbolic Execution API (key.core.symbolic_execution)
Proof Reference API (key.core.proof_references)
Proving (key.core)
Utilities (key.util)

# A Component of the KeY System

- The KeY System consists of well defined components
- Each component implements one or only a few clearly described responsibilities
- A component is a ready to use Eclipse Java Project consisting of:
  - `bin` // The compiled byte code
  - `src` // The source code
  - `lib` // The needed libraries
  - `resources` // Additional resources like images
  - `META-INF` // Jar definition files and services
  - `build.xml` // Ant script for deployment
- The separation of the KeY system into reusable components has the goals:
  - To support different use cases and languages at the same time (compile and runtime)
  - To avoid independent KeY branches

# Hierarchical Component Dependencies

- Transitive dependencies are not shown for simplicity
- Component with application logic | Component with unit tests



# Component Responsibilities

- ***key.util***: General utilities for Java: Can be used independent of KeY like immutable collections
- ***key.core***: User interface independent functionality to load source code, to perform proofs including rules, pretty printing and persistence
- ***key.core.testgen***: User interface independent functionality to generate counter examples and test cases
- ***key.core.symbolic_execution***: KeY's symbolic execution API independent from any user interface
- ***key.core.proof_references***: User interface independent functionality to compute proof references
- ***key.ui***: The typical Swing and console user interface of KeY including examples

# Design Guidelines

- Each component contains everything to fulfill its responsibility
  - Types of other components are not enriched for simplicity (e.g. to store additional Proof statistics)
  - General solutions are preferable (e.g. provide an arbitrary data Map to store statistics in)
- Types of other components can be used and extended (sub classing)
- Behavior of components can be modified via interfaces
  - EventListener to observe progress and state changes (e.g. `ProofDisposedListener` to observe `Proof#dispose()`)
  - Hooks via interfaces to modify behavior. Instances are configured by a configuration file and instantiated via `org.key_project.util.reflection.ClassLoaderUtil#loadServices(Class)` to avoid cyclic dependencies between components. (e.g. `ProofInitServiceUtil#getDefaultProfile(String)` to request the default `Profile` instance with a given name accessible via `DefaultProfileResolver` instances specified by `META-INF/services/de.uka.ilkd.key.proof.init.DefaultProfileResolver`)

# Separation between core and user interface

- Component *key.core* allows user interface independent proving, but
  - Results and progress need to be shown
  - User input might be required to complete a rule application
- The solution is a common API provided by *key.core*:
  - `UserInterfaceControl`: Allows to load source code, to instantiate new proofs or to register existing proofs and to access the `ProofControl`
  - `ProofControl`: Allows to list rules, to apply an individual rule, to start the auto mode and to run macros
  - `KeYEnvironment`: Provides static methods to load source code in a default user interface which is not shown to the user at all
- Component *key.core.example* shows how to programmatically verify all proof obligations of the source code.

# Implementation of the Swing user interface

- Component *key.ui* realizes the typical Swing user interface (`WindowUserInterfaceControl`) and the batch mode using the console (`ConsoleUserInterfaceControl`)
- The `KeYMediator` manages selected proof and node and provides other implementation specific behavior
  (e.g. freezing of the user interface)
  - The `KeYMediator` is **not** responsible to apply rules or to start the auto mode or macros. This is the responsibility of the `ProofControl`
- **Attention**: Do **not** add implementation specific functionality to the `UserInterfaceControl` or `ProofControl`!
  - May extend implementation classes `AbstractMediatorUserInterfaceControl` or `MediatorProofControl` instead.

# Future Work

- Profiles for different use cases (verification, information flow, …) and languages (Java, ABS, ...)