Glarimy Presentation on

Object Oriented

Design Patterns

Krishna Mohan Koyya

Proprietor & Principle Consultant

Glarimy Technology Services | Benguluru | Bharat

http://www.glarimy.com | @glarimy | krishna@glarimy.com

version 1.0 | 27th September 2014

Agenda

- **Design Patterns**
  - Object Orientation
  - Analysis to Design
  - Patterns
- **Creational Patterns**
  - Factory Method, Singleton, Prototype, Abstract Factory
- **Structural Patterns**
  - Adapter, Proxy, Facade
- **Behavioral Patterns**
  - Chain of Responsibility, Mediator
  - Observer, Visitor

The Classroom Protocol

- **Few Administrative Formalities**
  - Sign in the attendance sheet, if provided
  - Fill-in the feedback at the end of sessions, if provided
- **Derive Maximum Value**
  - Ask questions, any time
  - Participate in discussions and lab exercises
  - Avoid brining personal or office work to the class room
  - Stay away from internet while class in the session
- **Lets Learn Together**
  - Switch-off or mute personal phones
  - Leave/enter the class room without disturbing the class
  - Take phone calls outside the class room
  - Keep the class room clean and professional
  - No cross discussions
- **Reference Material**
  - Download from http://workshops.glarimy.com?patterns
  - No Print/CD. Protect Environment

Glarimy Technology Services

- **About Glarimy**
  - Established in 2008 and Registered in 2010
  - Based out of Benguluru, Bharat (Bangalore, India)
  - http://www.glarimy.com
- **Business Interests**
  - Technology Consulting
    - Architectural Appreciation and Review
    - Prototyping
  - Corporate Training
    - Software Design, Architecture and Processes
    - Web 2.0 Technologies
    - Standard and Enterprise Technologies
  - Weekend Public Workshops
    - One-day Executive Workshops on Saturdays
    - http://workshops.glarimy.com
- **Business Reach**
  - Geographies: Banguluru, Chennai, Pune, Hyderabad, Kochi and etc.,
  - Client Base: Sharp, GlobalDoc and LamResearch
  - Partner Clients: Robert Bosch, Samsung, Cisco, Vmware and etc.,

Krishna Mohan Koyya

- **Career spanning across 17 years**
  - 9 Years into Software Engineering with Cisco Systems, Wipro/Lucent and HP
  - 6 Years as Principle Consultant and Founder of Glarimy Technology Services
  - 1 Year as CEO at Sudhari IT Solutions, Bangalore
  - 1 Year as HoD at Sasi Institute of Technology and Engineering, Tadepalligudem, Andhra Pradesh
- **Proven Delivery**
  - Web 2.0: ExtJS, Dojo, Jquery Frameworks, AngularJS, HTML5, CSS3, XML
  - Enterprise Java: Spring, Hibernate, EJB, JSF, OSGi, Web Services, xUNIT family
  - Design Patterns, Architectural Patterns, TDD
- **Technology Expertise**
  - Distributed Systems and Network Management Systems
  - Object Oriented Systems Development
  - Infrastructure, Middleware and Web Layers with Java and Javascript
- **Academics**
  - M.Tech (Computer Science & Tech) from Andhra University, Visakhapatnam
  - BE (Electronics & Comm. Engg) from SRKR Engg. College, Bhimavaram

Agenda

- **Design Patterns**
  - Object Orientation
  - Analysis to Design
  - Patterns
- **Creational Patterns**
  - Factory Method, Singleton, Prototype, Abstract Factory
- **Structural Patterns**
  - Adapter, Proxy, Facade
- **Behavioral Patterns**
  - Chain of Responsibility, Mediator
  - Observer, Visitor

Software and Modelling

- **Software**
  - Set of Programs
    - Program is a set of instructions that can be executed by a computer
    - Code, Documentation, Configuration
    - Software sans any of the above becomes less useful
  - Automates existing business process
    - It [typically] never creates anything new for a business process
- **Model**
  - Abstracts a thing or a process or an idea
    - Presents only relevant detail, Leaves “extra” details
    - Always have a context and a purpose
    - No two models may be exactly same
- **Object Orientation**
  - Models existing business processes
    - In terms of objects and their interdependencies

Object Orientation

- **Abstraction**
  - One of the fundamental principles of Object Orientation
    - Presents only “required” detail about an object
    - A means to deal with the complexity
- **Encapsulation**
  - Wraps the object internals
    - Users look only at the utility
  - Presents abstraction
    - Outsiders can deal with whole rather than parts
    - Creation, destruction, usage - in terms of whole
    - Objects become reusable
- **Controls access to the internals**
  - Protects the internal details of the object
    - Security, by default
  - Provides interface for controlled access
    - Public members for every one and private members for no one

Classes

- **Blue Prints**
  - Describes the structure & behaviour of objects
  - Do not do anything
    - They are not runtime elements
    - Only objects do, not the blue prints
- **Three Parts**
  - Attributes & Relationships
    - Attributes describe the static internal picture
    - Relationships describe dependencies among objects
  - Behaviour
    - Describes the dynamic picture
- **Class Level Data**
  - Specific to the given class
    - Data is available even before object is created
    - Shared among all object of the class

Objects

- **Represent real world objects/ideas**
  - An object oriented system is just a collection of objects
    - Runtime elements in the system
  - Represent tangibles as well as intangibles
    - Things, Processes, concepts
- **Instances of classes**
  - Each object belongs to a class
  - All class level data is available to each of the objects
  - A class can be realized into any number of objects
- **Object level data**
  - Specific to the given instance of the class
  - Not shared among objects
  - Data is available only with the specific object

Class Definition

- **Attributes**
  - Characteristics
    - Describes the inherent structure
  - Inseparable
    - Every object should have a value for the attribute
    - An object sans the defined attribute is not possible
  - Granular
    - Generally very granular in nature
    - May not be described using other structures
- **Relations**
  - Among objects
    - Cardinality: One-to-One, One-to-Many, Many-to-One, Many-to-Many
    - Role: Each object has a purpose
    - Navigation: At least one of them ‘knows’ the other
    - Type: Dependency, Association, Aggregation and Composition

Class Definition

- **State**
  - Snapshot of set of values of attributes & relations
  - Changes if any of these values change
    - Only object changes states, not class
- **Behaviour**
  - The way an object changes its state
  - Operation
    - Interface to invoke the behaviour
    - State better not be changed directly accessing it
  - Method
    - An implementation of an operation
    - An operation can be carried in more than one methods - Overloading
- **Message**
  - A means to invoke operation on (another) object

Inheritance

- **Deals with hierarchical complexity**
  - Reuses an existing class and extends it further
    - Specializes the general class
    - It's a relation of specialization between parent and child
- **It's an "IS-A" relation**
  - Every child class "IS-A" a parent class, in a way
    - Vice-versa is not true
    - Also not true between two child classes of the same parent
  - **Parent Classes & Child Classes**
    - Child inherits everything from the parent
      - Parent can stop one/some/all of its members being inherited
      - All protected members are meant only for the self & children
    - Child can override parent's behaviour
      - Parent can stop one/some/all of its methods being overridden
    - Child can extend the parent
      - By adding its own attributes/associations/behavior
      - Parent can stop being extended, of course
    - A child extends more than one parent

Interfaces and Polymorphism

- **Classic Distinction**
  - What & How
    - “What” deals with “requirement”
    - “How” deals with “implementation”
- **Classes, Objects and Interfaces**
  - Class specifies both “what” & “how”
  - Interface specifies only “what”
    - Interface can not do anything, its abstract
    - A class needs to realize this interface by providing “how” specs
    - More than one class can realize the same interface
    - Each class provides different “how” to the same “what”
  - Object exhibits that specification at run time
- **Polymorphism**
  - Users deals with interfaces rather than actual objects
    - Different implementations are supplied at different contexts
    - The behaviour is not known till runtime

Object Oriented Analysis

- **Understand the big picture**
  - Don’t look for trees, find the forest
  - Identify the scope and boundaries of the system
- **Generate User Stories**
  - Interact with the real-users of the system, not just customer
    - Develop Use Cases
    - Identify the external actors, triggers, expected outcomes & exceptions.
- **Model the existing business process**
  - Find various parties involved in the show
    - Identify all nouns
      - They are candidate objects or attributes or relations
    - Identify all verbs
      - They are candidate behaviour interfaces (operations)
    - Develop classes
      - Bind the attributes & behaviour to appropriate objects

Modeling with UML

- **UML**
  - A visual modelling language
    - Offers syntax and semantics
  - Roots are in software development
    - Applicable to any field
  - Not a process
    - Used as part of a process
- **Bit of History**
  - Object Orientation and Three Amigos
    - Booch - Booch Methodology
    - Rumbaugh - Object Modelling Technology
    - Jacobson - Objectory
  - Unified
    - 1994: Started in 1994 with Rational
    - 1997: UML 1.0 adoption by Object Management Group (OMG)

UML Diagrams

- **Structural Diagrams**
  - Class and Object Diagrams
    - Presents the structure
  - Package Diagrams
    - Groups classes/objects logically
  - Component Diagrams
    - Groups classes/objects functionally
  - Deployment Diagrams
    - Presents distribution of software onto hardware
- **Behavioural Diagrams**
  - Use Case Diagrams
    - Presents system from external point of view
  - Activity and Sequence Diagrams
    - Models a specific flow
  - State Machine Diagrams

Use Case Analysis

- **Use Case Analysis**
  - An approach to requirement analysis
    - Customer & Analyst participate
  - Comes out with User Stories and Use Case Diagrams
    - Identifies system boundaries
    - Finalizes the requirement scope
    - Focus is on “what needs to be delivered”
    - Not concerned about “How the functionality will be achieved”
- **Deliverables**
  - Use Cases
    - Feature of the system from external point of view
    - Helps in putting together a requirements traceability matrix
  - Use case Diagrams
  - Helps in putting together the requirements specifications

Use Case Diagram

- **Collection of Use Cases, Actors and Subject**
- **Subject**
  - Whose functionality is being analyzed
  - Typically of a process/subsystem/component/class and etc
- **Actor**
  - Uses the use case or receives the use case results
    - External to the subject, not part of it
    - May not map to real users, but to their roles
    - Can inherit other actors
- **Use Case**
  - A feature of the system from the external user point of view
    - Must be initiated by an actor
  - When it is completed
    - Desired functionality must have been performed or an error has been thrown

Class Diagram

- **Presents classes and their relations, statically**
- **Class**
  - Three compartments: name, attributes, operations
- **Attributes**
  - Primitive types (Inline attributes)
  - References (Relationships)
  - Inline attribute
    - visibility /name :type multiplicity default {property strings & constraints}
    - visibility: +, -, #, ~
    - multiplicity: [lower...upper], [5], [1..5], [*], nothing is one.
  - Derived Attributes
    - Prepend with /
    - Computed based on other parts of the class, typically read-only
  - Static Attributes
    - Underline

Class Diagram

- **Operations**
  - To specify how to invoke behaviour
  - op:= visibility name (params): return-type {properties}
    - params:= direction param: type [multiplicity] = default
  - Exceptions
- **Relations**
  - Multiplicity: Cardinality (How many are participating)
  - Navigability: Directivity (How is aware of whom)
  - Role: How its known to the other
  - Nature of Relation:
    - Dependency: weakest, uses, dashed line
    - Association: has-a, solid line
    - Aggregation: owns-a, solid line with empty diamond on owner
    - Composition: is-part-of, not shared, all others, filled-diamond on owner

Class Diagram

- **Generalization**
  - is-a
  - Solid-line with triangle head on parent
- **Abstract Class**
  - Class name is in italics
  - Some or all or none of the operations are abstract
  - Can't be instantiated
- **Interface**
  - Pure abstract
  - Need not to have operations
  - Stereotype <<interface>>
- **Abstract classes and Interfaces are realized**
  - Dashed inheritance

Sequence Diagram

- **Participants, Lifelines and Triggers**
  - <<create>>, <<destroy>>, local variables, <<self>>, filled arrows
  - Message:= attribute=signal/opName(args):return value
- **Creation & Deletion**
  - Object creation using arrowed dashed line with <<create>>
  - Object deletion using <<destroy>>
- **Messages**
  - Asynchronous message as open arrow
  - Return values using arrowed dashed line
  - Found message from unknown sender using filled circle
  - Lost message to unknown destination using filled circle
- **Execution occurrences**
  - Rectangles on the lifeline

Analysis to Design

- **Design Considerations**
  - To meet both functional and non-functional expectations
  - Enterprise Quality Expectations
    - Code reusability, Reliability, Scalability
    - Robustness, Security,
    - Efficiency, Usability and etc.
- **Typical Considerations**
  - Modularity
    - For Maintainability and Reusability
  - Loose-coupling
    - For localizing change impact
  - High-cohesion
    - Single Responsibility
  - Information-hiding

Analysis to Design

- **Separate Classes**
  - Identify the business classes from the domain classes
    - Business class have some behaviour
    - Domain class holds only state
- **Introduce Interfaces**
  - Create an interface for each of the business classes
    - Covert the business class as a realization of the interface
    - Throw an exception from each of the operation
    - Have an application-wide or layer-wide parent exception class
      - Inherit specific exceptions from the application exception
- **Identify Specific Design Considerations**
  - Design considerations often in conflict with each other
    - Example: Performance Vs Reusability
  - Prioritize the considerations
- **Apply appropriate design patterns**

GoF Design Patterns

- **Proven solutions to recurring design problems**
  - The famous Gang of Four (GoF) published 27 patterns
- **Three Categories**
  - Creational Patterns
    - Solutions for problems associated with object creation
      - Factory Method, Abstract Factory, Prototype, Singleton, Builder, Bridge
  - Structural Patterns
    - Solutions for problems associated with object relationships
      - Adapter, Flyweight, Proxy, Composite, Façade, Decorator, Template
  - Behavioural Patterns
    - Solutions for problems associated with object interactions
      - CoR, Iterator, Mediator, Observer, Command, Strategy, Visitor, Memento, Interpreter
- **Derived Patterns**
  - Solves design problems of a specific domain/layer/technology
  - Adaptation of GoF patterns to the environment

Got Design Patterns

- **What are design patterns**
  - A design pattern is a proven solution for a commonly found design problem
  - No pattern can be implemented without the context.
  - Gang of Four published a catalogue of patterns in three categories
    - Creational, Structural and Behavior
- **The Practice of Design**
  - Have interfaces for the business classes, unless there is a strong reason for not doing so.
    - *You may do the same for domain classes as well.*
  - Always look for high-cohesion and loose-coupling
    - *Use polymorphism and code-against-interfaces mantra.*
- **Applying Patterns**
  - Only once the basic design guidelines are taken care
  - Using a wrong pattern will have serious consequences. Never force a pattern.
  - Patterns always have to be customized for the given system/app/technology/domain.

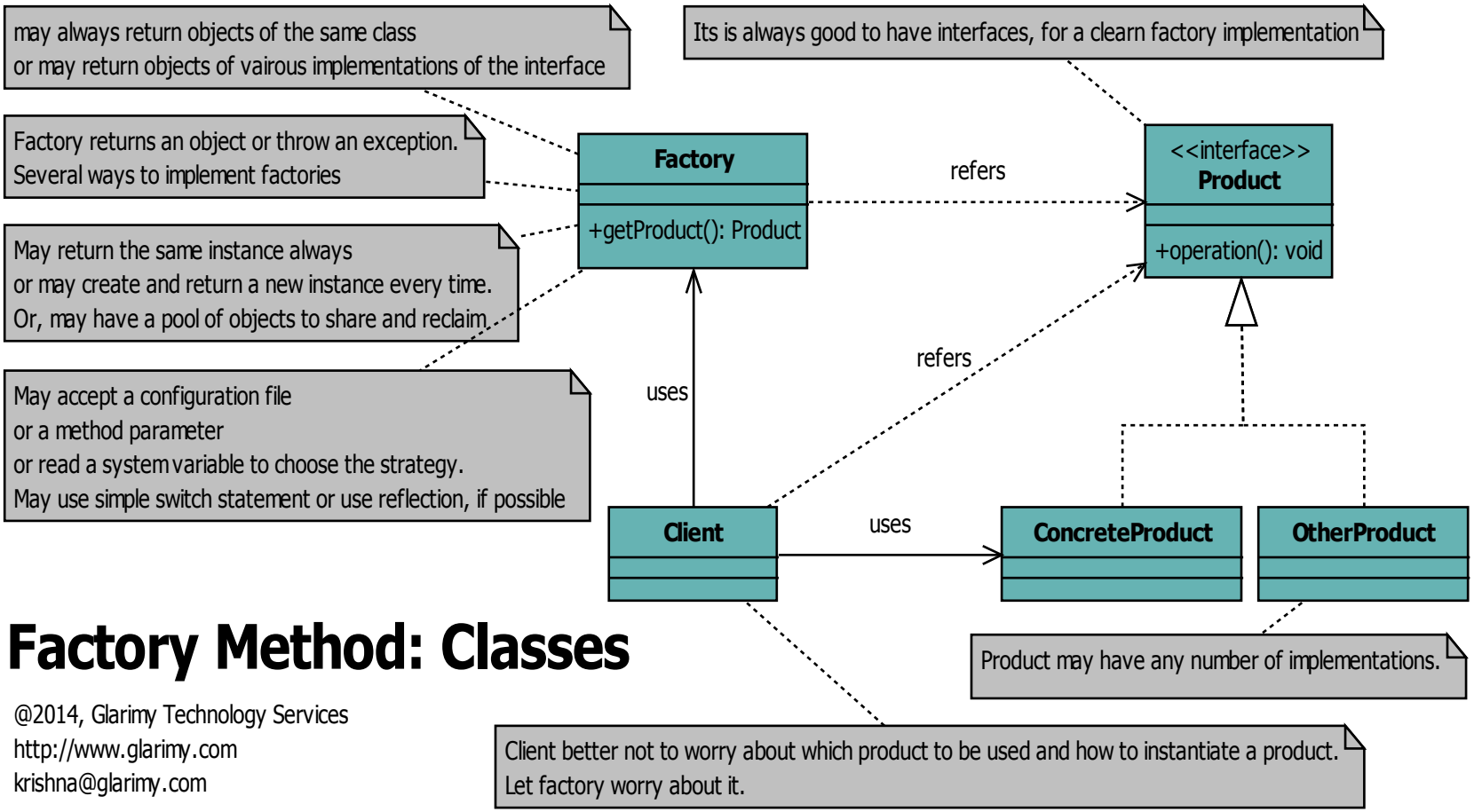
Agenda

- **Design Patterns**
  - Object Orientation
  - Analysis to Design
  - Patterns
- **Creational Patterns**
  - Factory Method, Singleton, Prototype, Abstract Factory
- **Structural Patterns**
  - Adapter, Proxy, Facade
- **Behavioral Patterns**
  - Chain of Responsibility, Mediator
  - Observer, Visitor

Factory Pattern

- **Intent**
  - Separating responsibilities of object creation from object usage
- **Description**
  - Scenario: Most of the business classes implement an interface. When there is a chance for more than once implementation for an interface, choosing the implementation becomes a responsibility. Factory takes that responsibility.
  - Scenario: An application may want to have a pool of objects which they want to share. The size of pool may grow or reduce depends on the load. A factory takes care of such a pool management.
- **Implementation**
  - factory may create a new instance every time some one ask for an instance
  - A factory may create a pool of instances upfront, so quickly gives the objects to the clients.
  - A factory may create just give reference of an object but not the life cycle management of the object.
  - A factory may share the same object among several clients.
- **Relations**
  - Factory may use prototype, strategy patterns

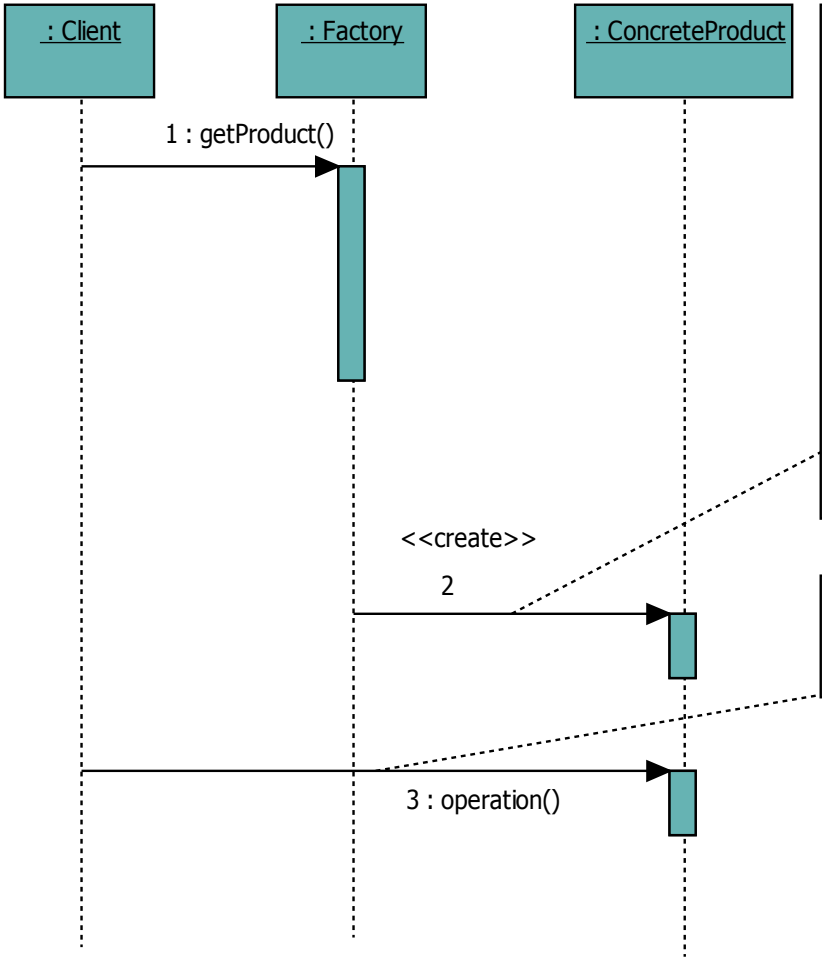
Factory Pattern



Factory Method: Classes

@2014, Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Factory Pattern



May always return objects of the same class
or may return objects of vairous implementations of the interface

Factory returns an object or throw an exception.
Several ways to implement factories

May return the same instance always
or may create and return a new instance every time.
Or, may have a pool of objects to share and reclaim

May accept a configuration file
or a method parameter
or read a system variable to choose the strategy.
May use simple switch statement or use reflection, if possible

Client better not to worry about
which product to be used and how to instantiate a product.
Let factory worry about it.

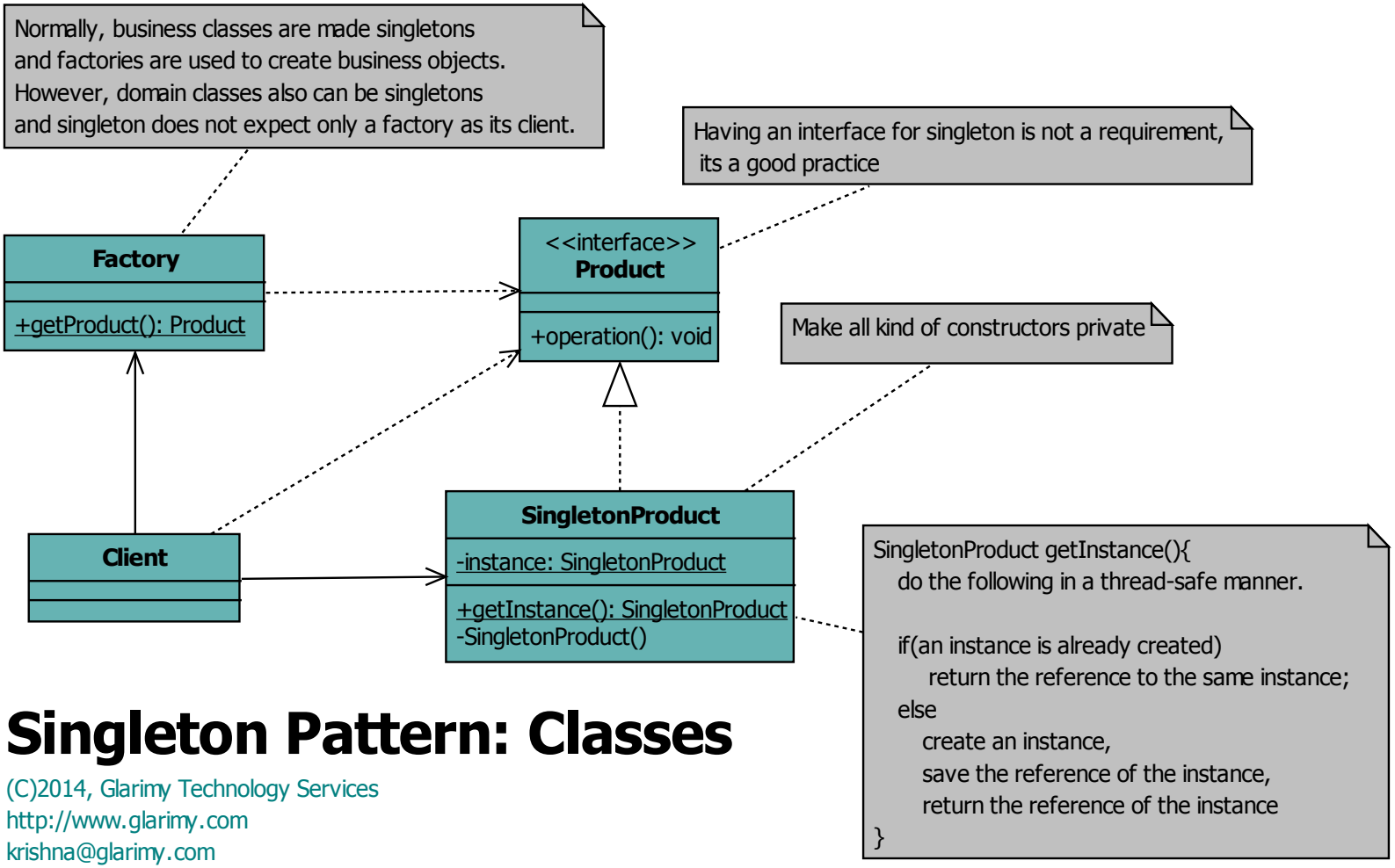
Factory Method: Interactions

@2014, Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Singleton Pattern

- **Intent**
  - To limit the maximum number of objects of a class to only one.
- **Description**
  - Scenario: An application can not afford to have more than once instance of a class and at the same time it is not a mandatory object to be available at the start up of the application.
- **Implementation**
  - Make the constructor private and provide a static factory method to create the instance under controlled conditions
  - If language supports cloning or copy-constructors, take additional care.
  - The factory method may be synchronized in order to be thread-safe.

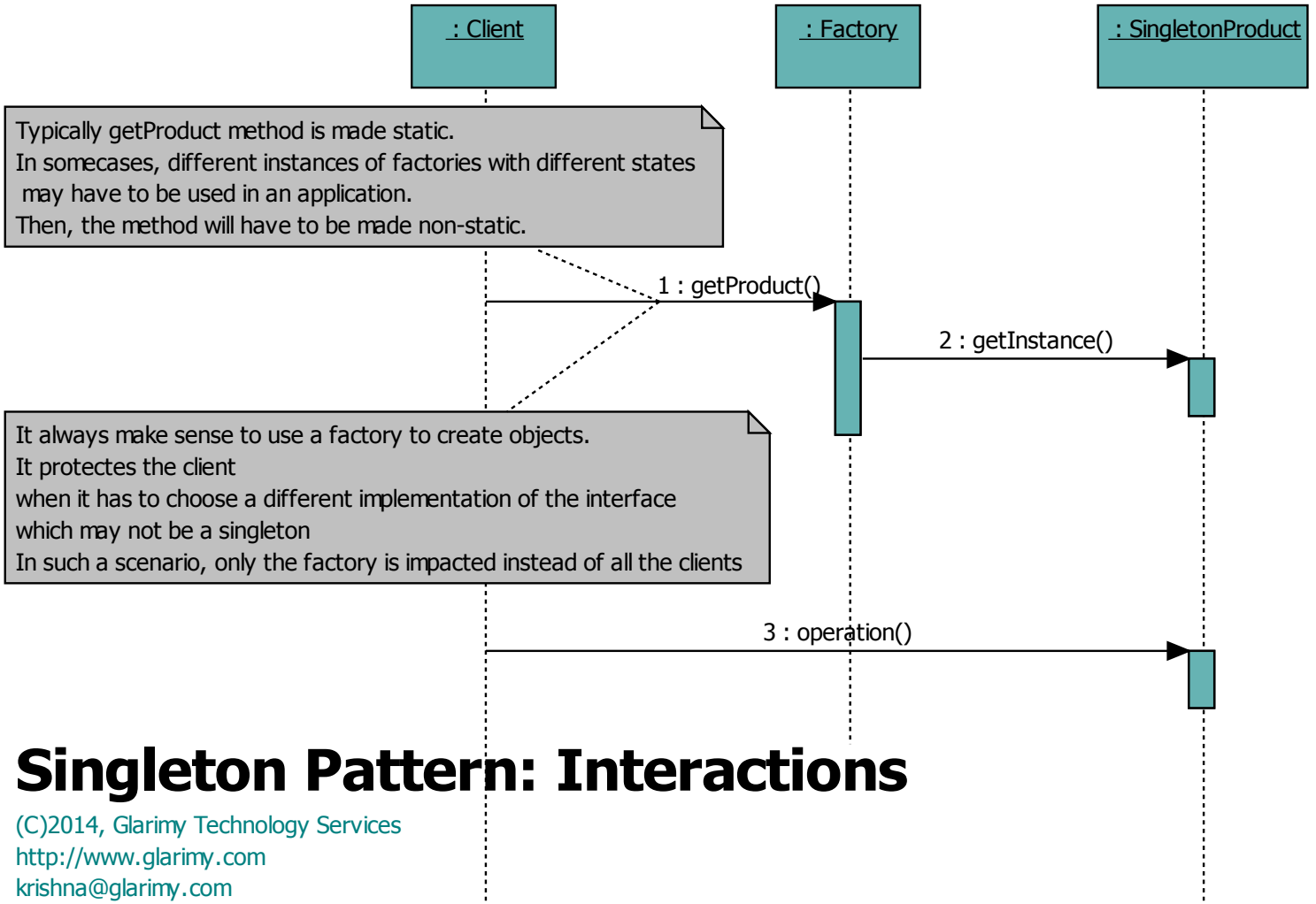
Singleton Pattern



Singleton Pattern: Classes

(C)2014, Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Singleton Pattern



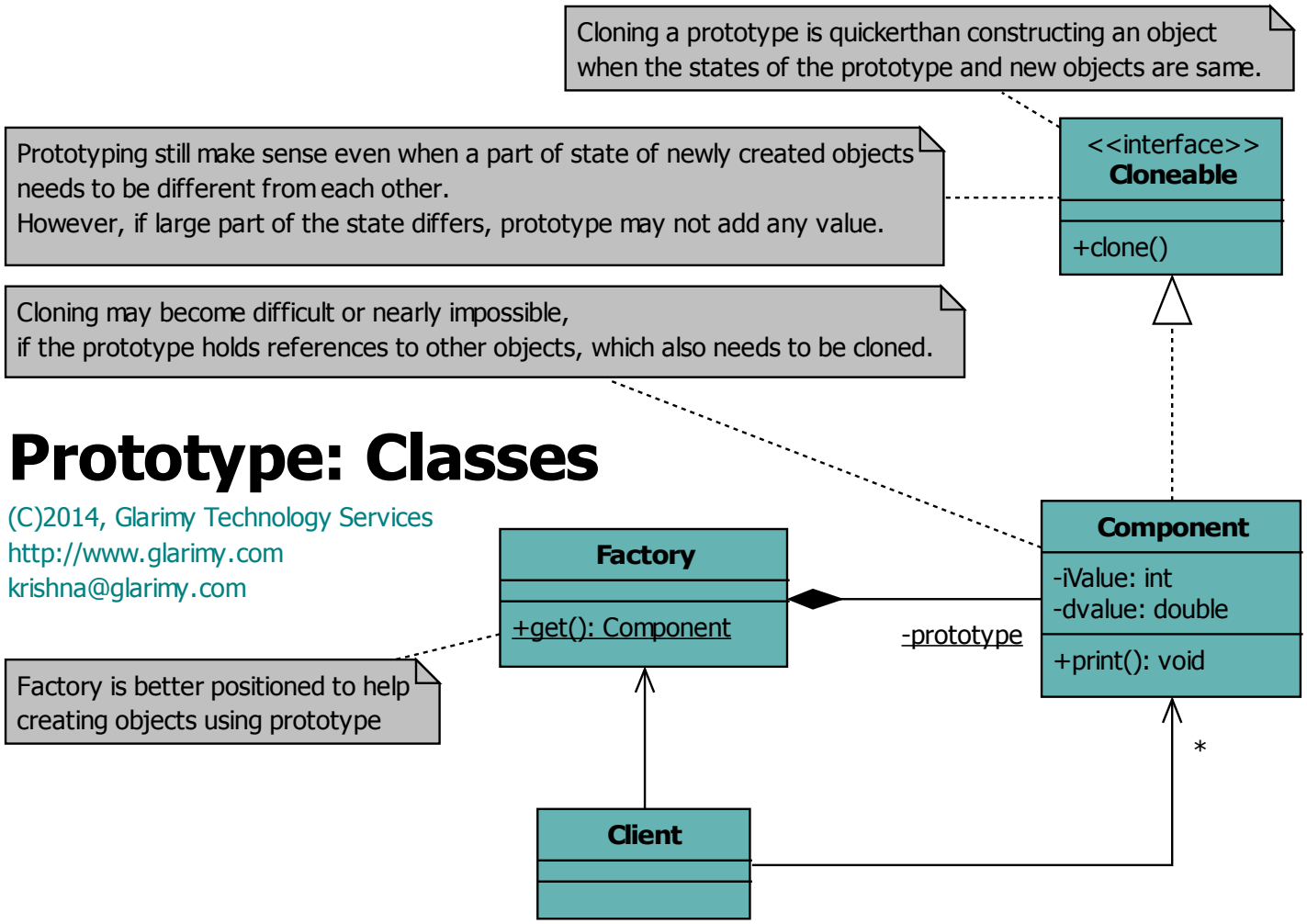
Singleton Pattern: Interactions

(C)2014, Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

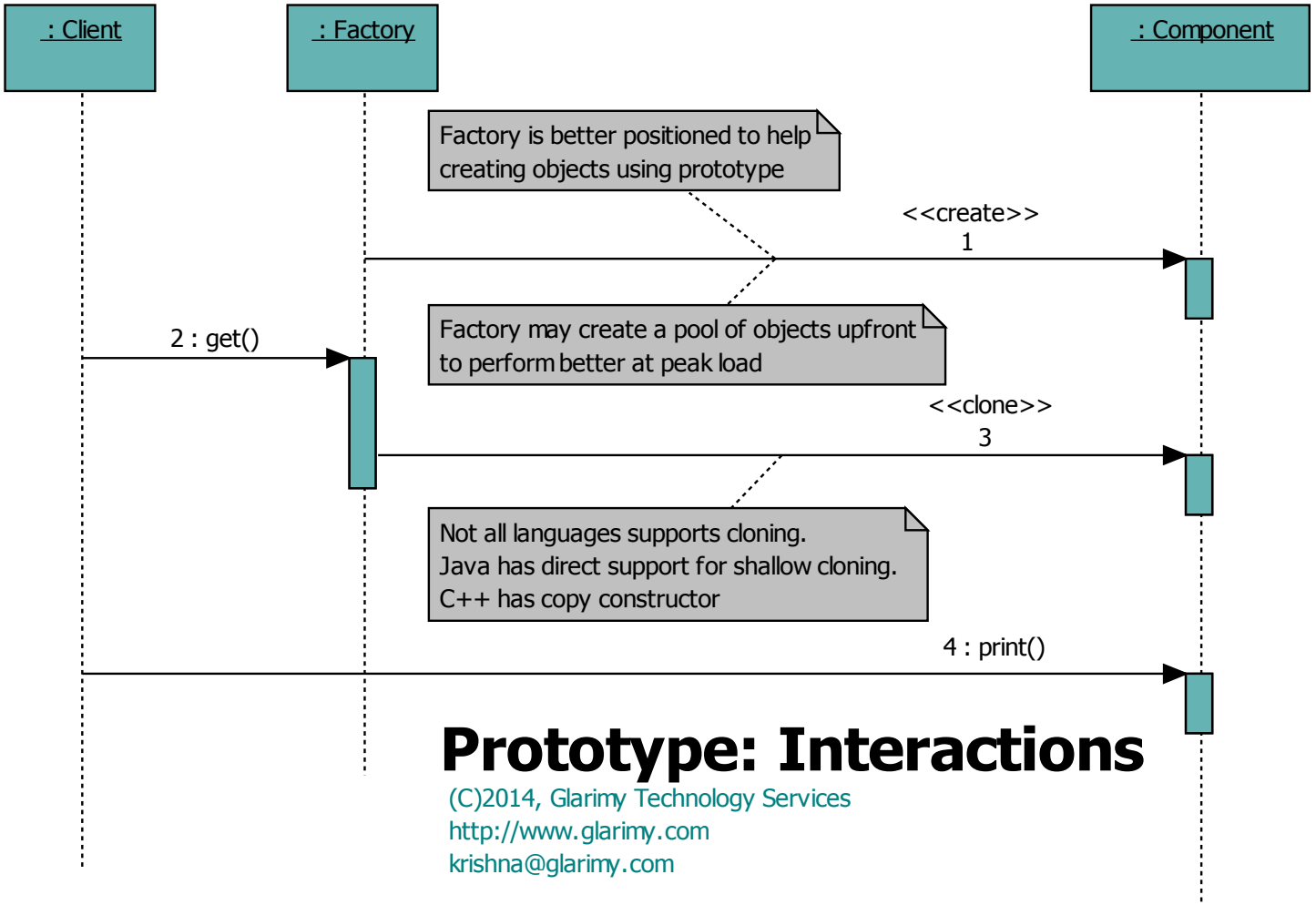
Prototype Pattern

- **Intent**
  - To speed up creating a large pool of objects that share common initial state..
- **Description**
  - Scenario: A gaming application need to create large number of objects like stars, arrows and etc., Given the scenario, all these objects starts with more or less same state like size, color, shape and etc., but have independent life. These objects need to be created as quickly as possible in large number.
- **Implementation**
  - Make the class of these objects as cloneable or provide a copy constructor depending on the language.
  - Create one instance of this cloneable class with that common initial state. This object is called prototype.
  - Then clone this prototype to churn out any number of objects thereafter. They all share the same state initially, but they are truly different objects. So they can be used independent of each other.
  - The real benefit of this pattern is to increase the performance as cloning is faster than constructing an object.
  - Need to take the constraints in using deep-cloning into consideration.

Prototype Pattern



Prototype Pattern



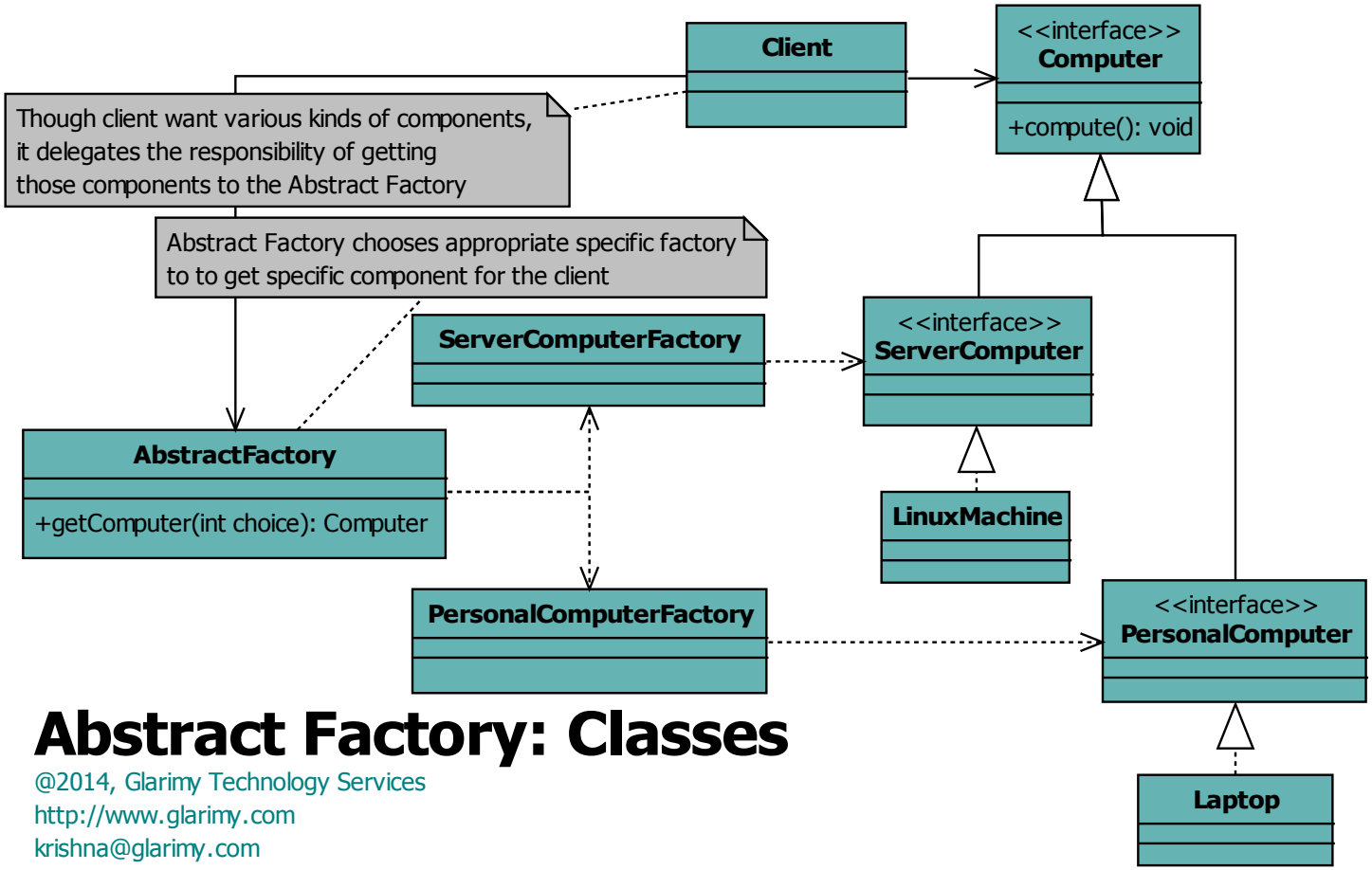
Prototype: Interactions

(C)2014, Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Abstract Factory Pattern

- **Intent**
  - To provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Description**
  - Scenario: A system should be built with various constituent objects which belong to same family. Each kind of those objects can be created by factories meant for them. However, responsibility of choosing correct factory for a given object should not be given to the system. Instead an abstract factory takes care of it.
- **Implementation**
  - Represent the object family as an interface and extend it to sub-interfaces for creating object class implementations.
  - Have a factory for creating object instances from each of the sub-interface implementations.
  - Finally, create a factory, the abstract factory, with the intelligence to choose the appropriate factory while creating objects.
  - Abstract Factory is also a factory. Use configuration, reflection, switch-logic and etc., to build the factory method of the Abstract Factory.

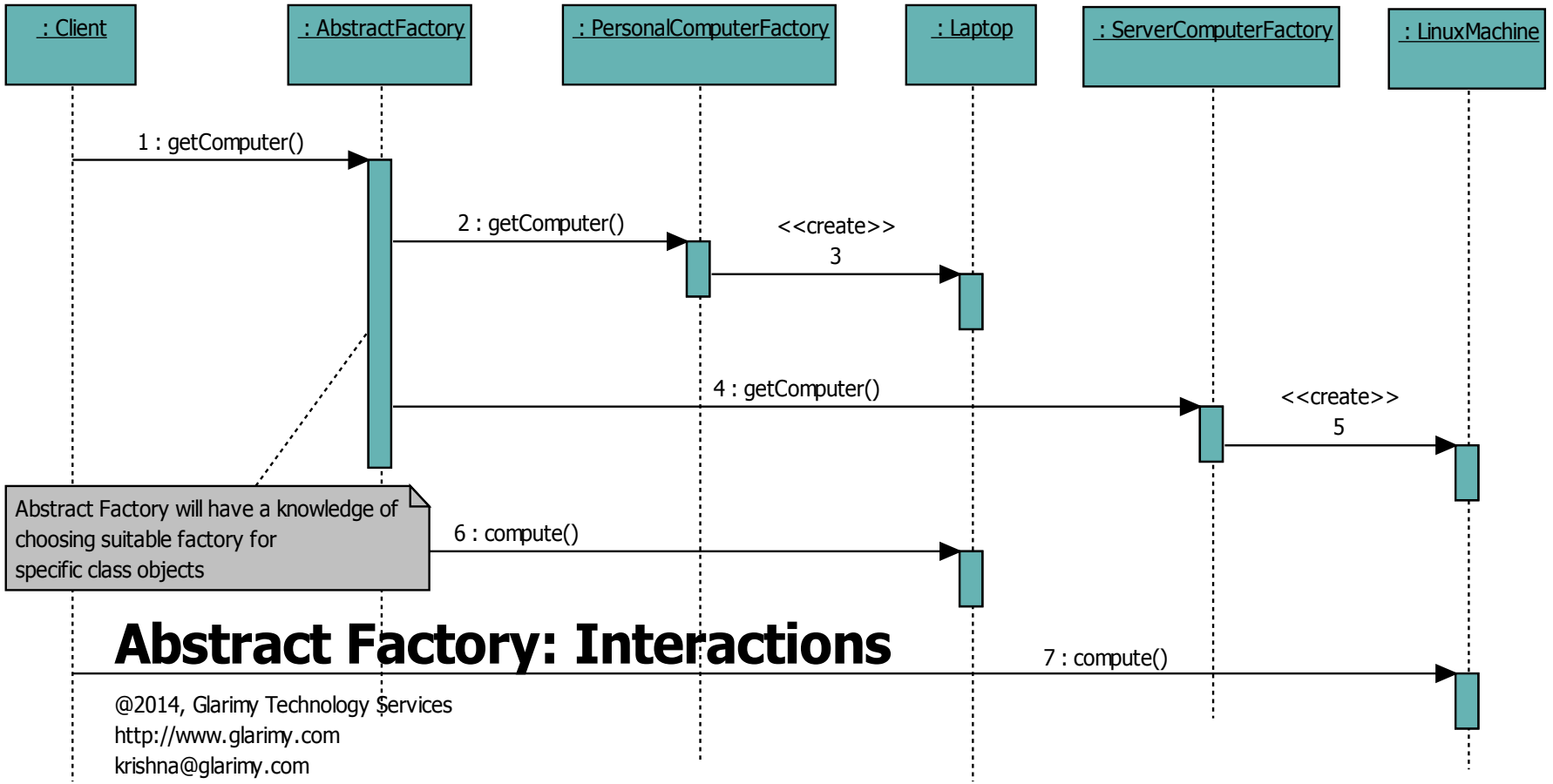
Abstract Factory Pattern



Abstract Factory: Classes

@2014, Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Abstract Factory Pattern



Abstract Factory: Interactions

@2014, Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

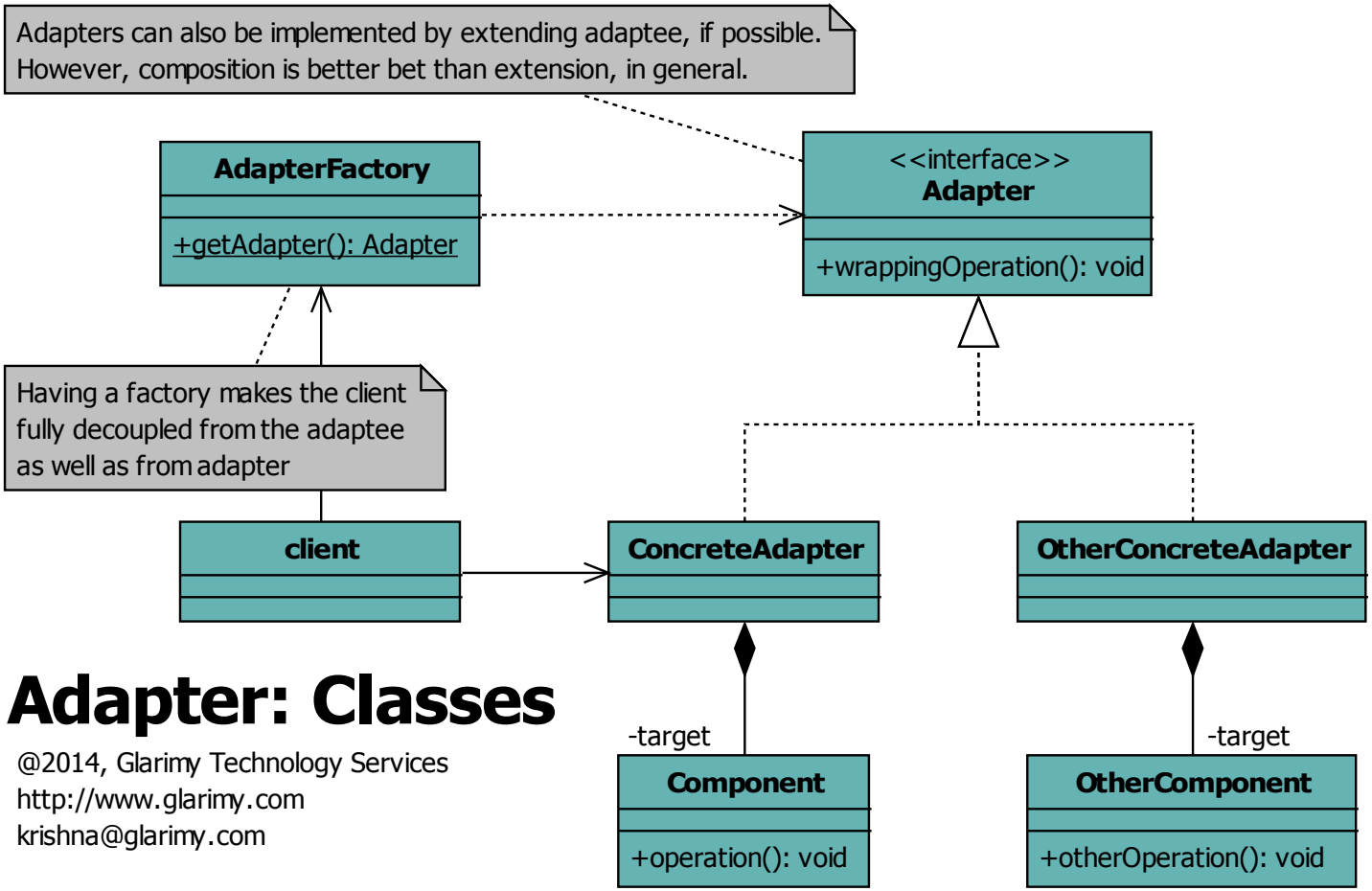
Agenda

- **Design Patterns**
  - Object Orientation
  - Analysis to Design
  - Patterns
- **Creational Patterns**
  - Factory Method, Singleton, Prototype, Abstract Factory
- **Structural Patterns**
  - Adapter, Proxy, Facade
- **Behavioral Patterns**
  - Chain of Responsibility, Mediator
  - Observer, Visitor

Adapter/Wrapper Pattern

- **Intent**
  - To provide a known interface for an unknown component
- **Description**
  - Scenario: A third-party component may have an interface which is not in the way that an application is expecting. A wrapper provides the known interface to the application while it internally translates the calls to 3rd party component.
- **Implementation**
  - A wrapper may compose the adaptee and forwards the calls from clients after due translation.
  - A wrapper may inherit the adaptee and forwards (upward to the super class) the calls from clients after due translation
  - A wrapper may have its own interface so that various concrete wrappers may wrap different adaptees.
- **Relations**
  - A factory may churn-out the wrappers to its clients

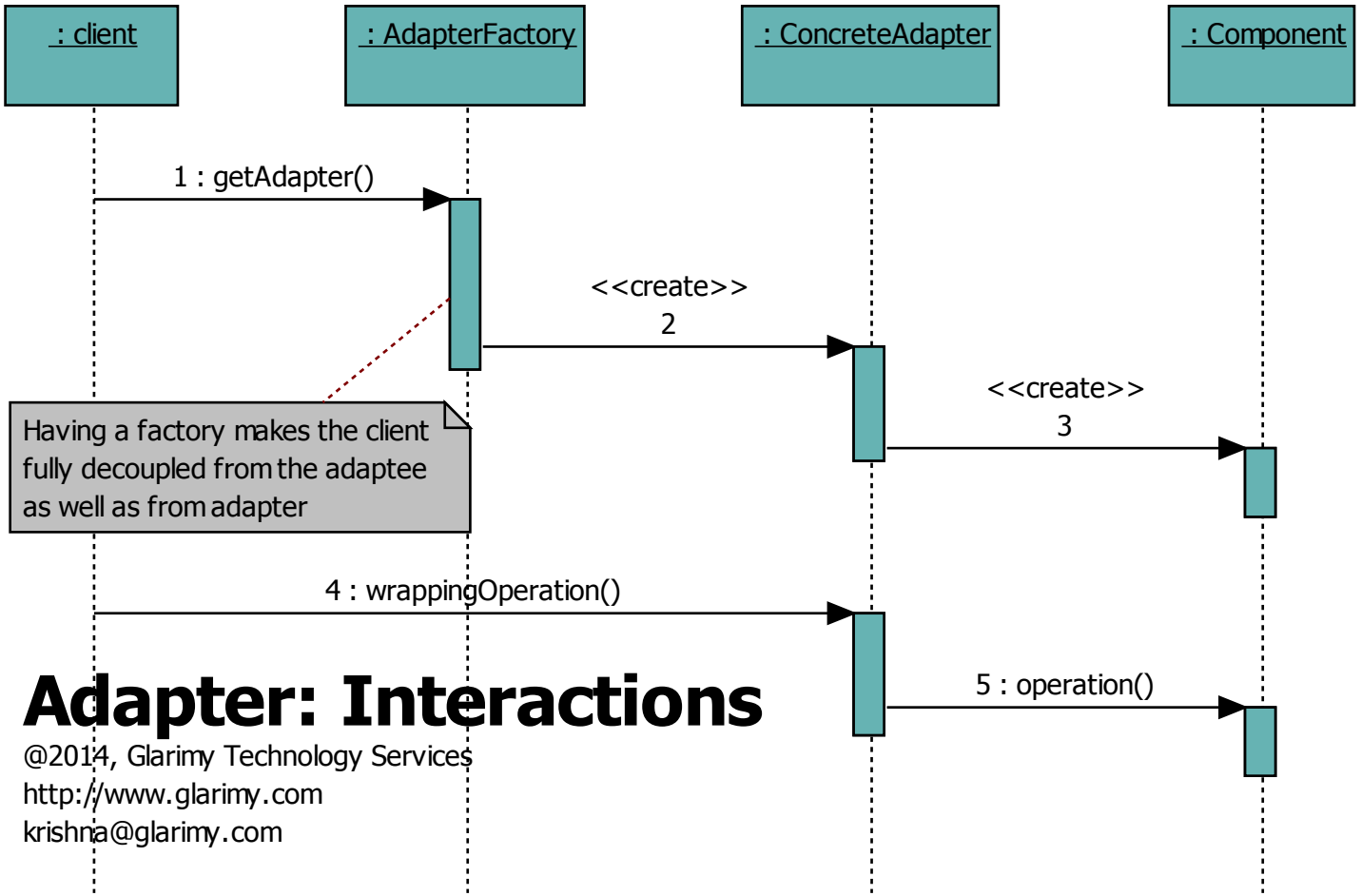
Adapter/Wrapper Pattern



Adapter: Classes

@2014, Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Adapter/Wrapper Pattern

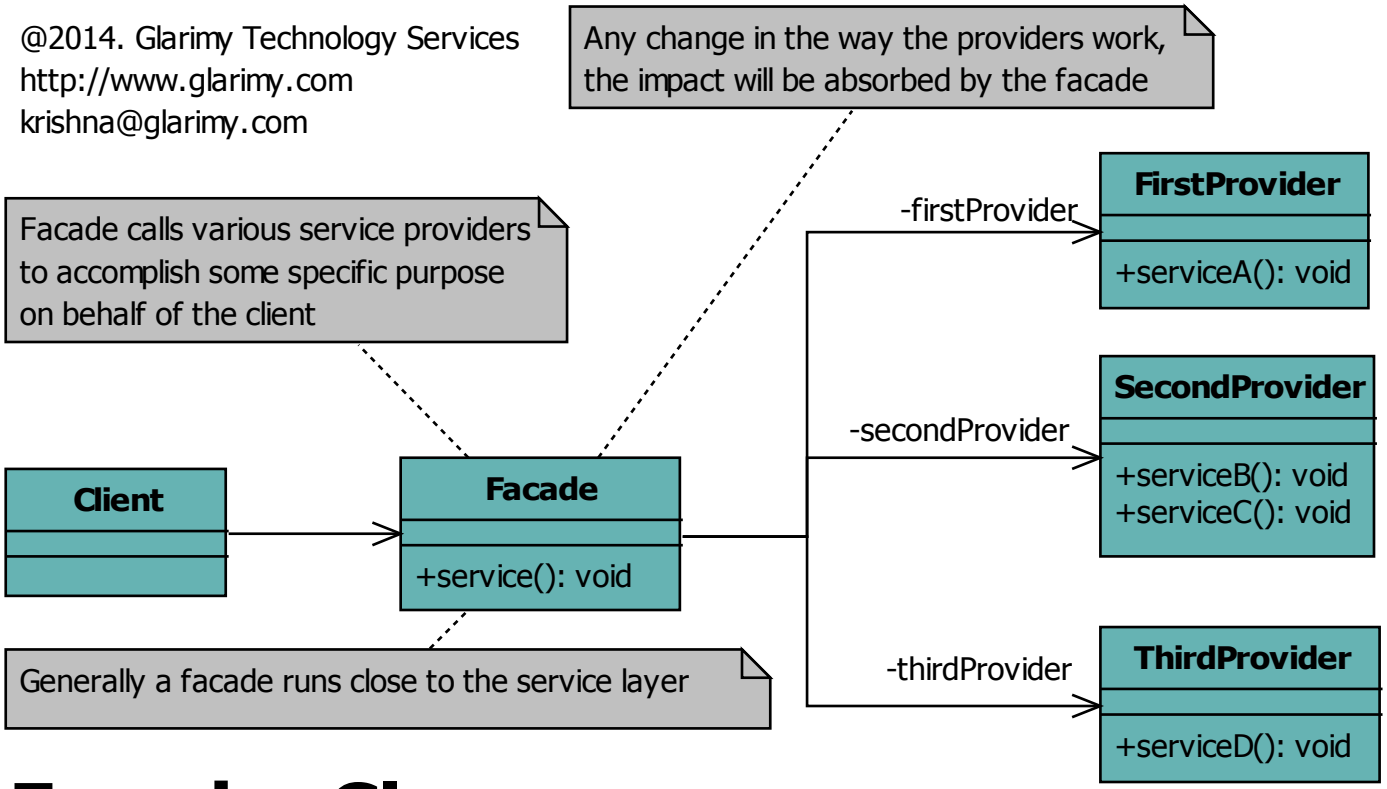


Façade Pattern

- **Intent**
  - To hide internal business logic from the clients
- **Description**
  - Scenario: When a client is forced to make calls to several components on the service layer, the impact on the client is more whenever the process of calling these components change. A façade hides such a process so that the client only work with the façade.
- **Implementation**
  - A façade may be simply a method of a class which internally calls various private methods of the class.
  - A façade may a class that holds references to various other components and orchestrate the interactions among them upon a client request.
  - Any change in the orchestration will have no impact on the client.
- **Relations**
  - A façade may take the shape of an observer or a mediator. Or it may simply turn out to be a wrapper or a proxy under special cases.

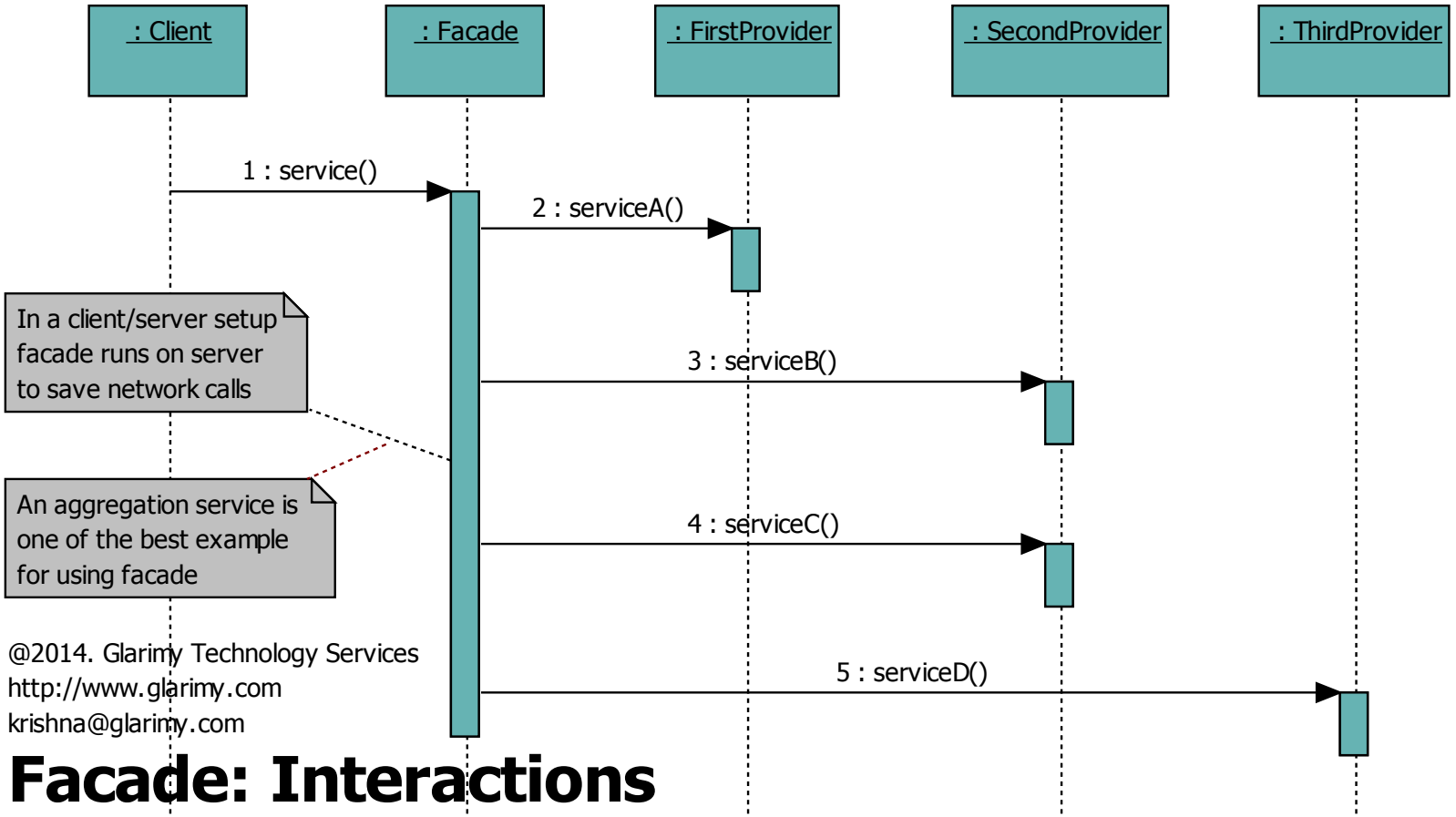
Façade Pattern

@2014. Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com



Facade: Classes

Façade Pattern

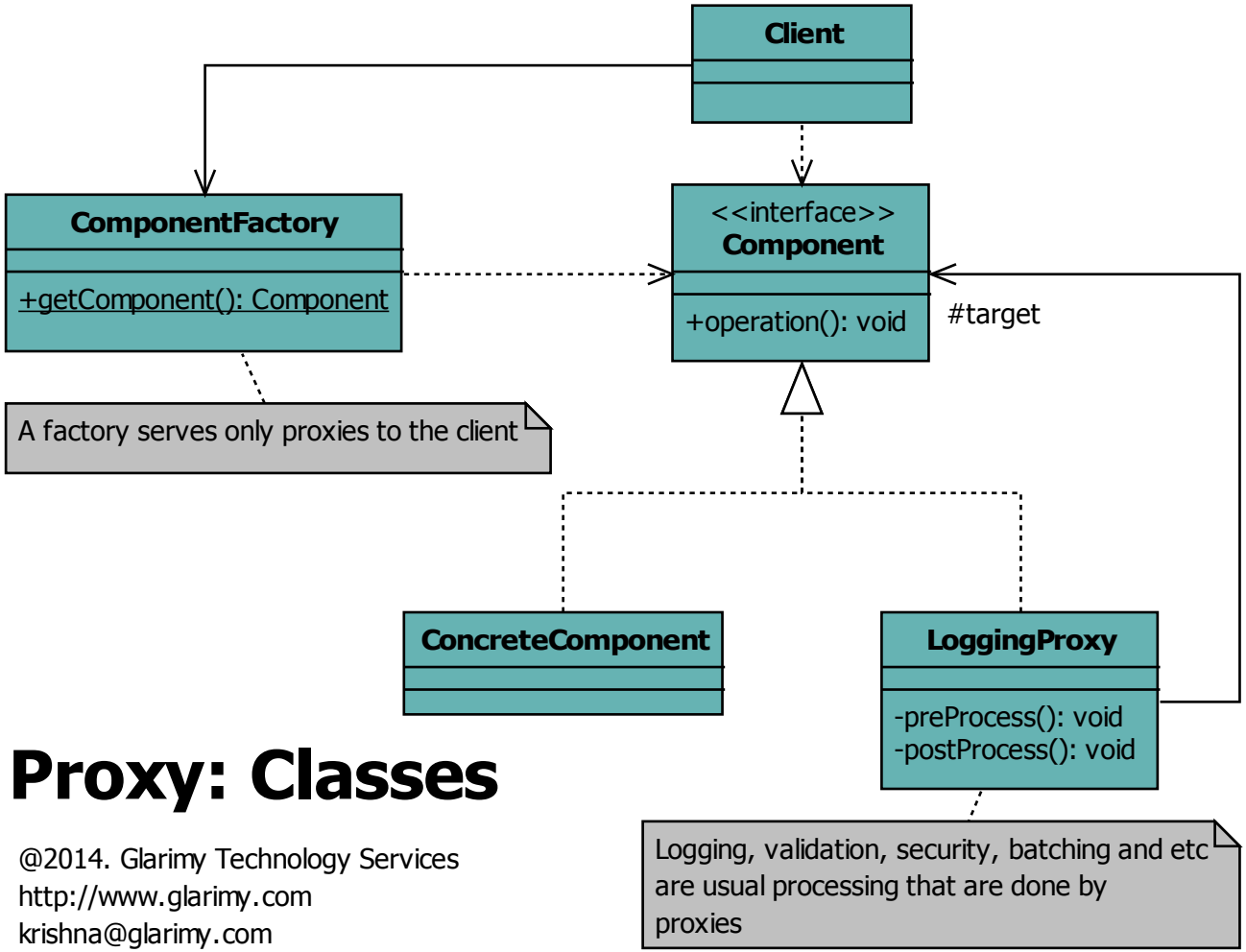


Facade: Interactions

Proxy Pattern

- **Intent**
  - To delegate the pre and post processing responsibilities from the client, transparently.
- **Description**
  - Scenario: In a client/sever applications, the client will have to deal with networking and protocols in order to talk to the business object. A proxy can take care of such a responsibility.
  - Scenario: Validation, Transaction management, Authorization, Logging and etc., are not really business concerns rather secondary concerns which can not be mandated on either business class or client for the sake of reusability. Proxy helps in all such optional pre and post processing scenarios.
  - Scenario: In order to reduce the load on server, a proxy may be employed to batch the calls from client.
- **Implementation**
  - A proxy may implement the same interface of the business class and hold reference to business object.
  - A proxy may simply extends the business class.
- **Relations**
  - Factories may be used to choose the proxies. Proxies may act as facades.

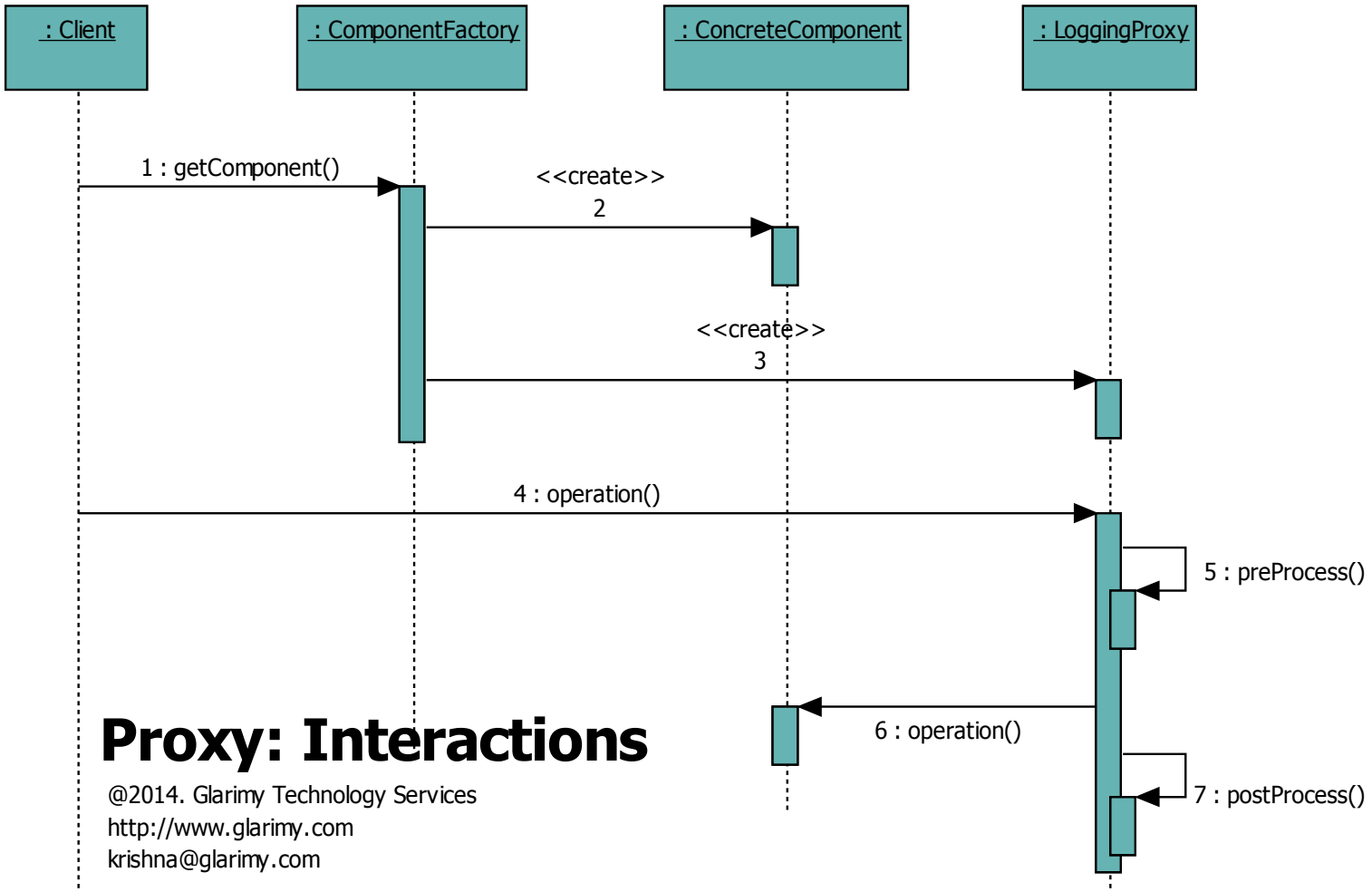
Proxy Pattern



Proxy: Classes

@2014. Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Proxy Pattern



Proxy: Interactions

@2014. Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

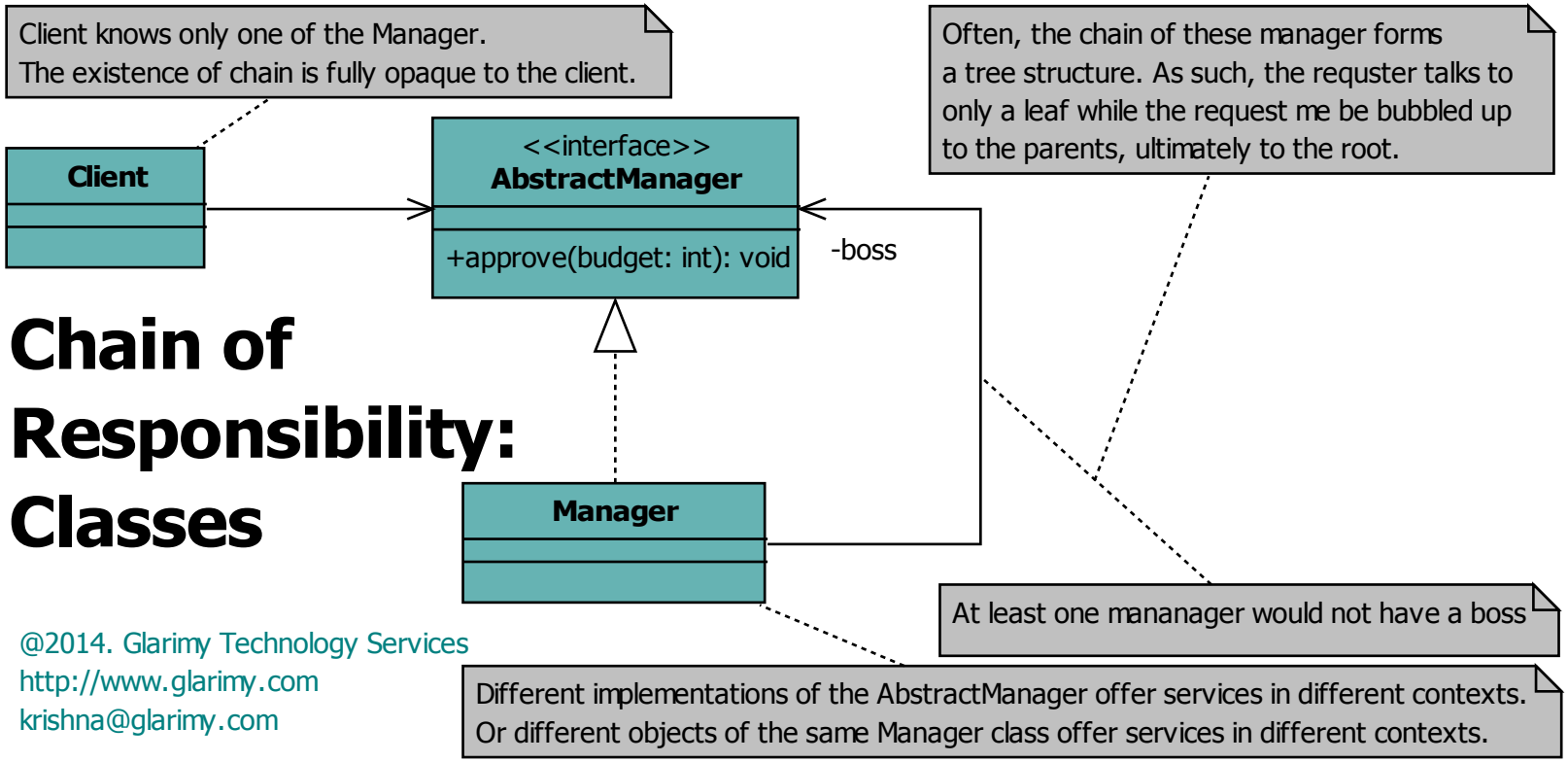
Agenda

- **Design Patterns**
  - Object Orientation
  - Analysis to Design
  - Patterns
- **Creational Patterns**
  - Factory Method, Singleton, Prototype, Abstract Factory
- **Structural Patterns**
  - Adapter, Proxy, Facade
- **Behavioral Patterns**
  - Chain of Responsibility, Mediator
  - Observer, Visitor

Chain of Responsibility Pattern

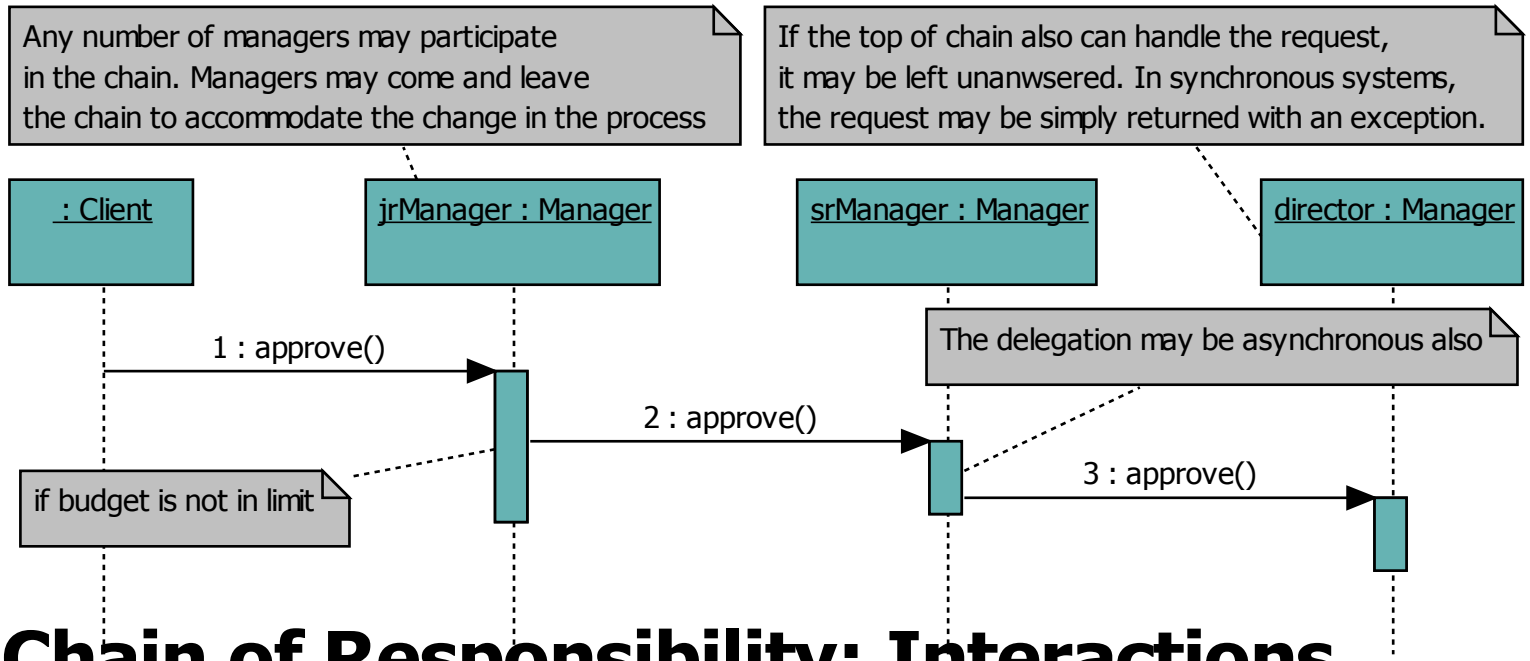
- **Intent**
  - To enable communication in a hierarchical chain in a such a way that a requester would get service without knowing the provider.
- **Description**
  - Scenario: A system consists of several service objects. They all provide similar service, but based on different contexts. For example, different managers approve budgets of different magnitudes. Different counselors offers help based on the kinds of the problems. If the requester has to know all the service objects and the context of the services, it becomes very tightly coupled. Instead, let the requester always approach a first in a chain of several service objects. The service objects either offer the service or delegates the request to the next in chain.
- **Implementation**
  - All the service objects implement the same interface. And they are connected in a such a way that every service object known its successors.
  - Each service object either handles a request fully or delegates to its successor.
  - Any new service can be included by introducing a new object to the chain.
  - Let the client knows the first in the chain.
  - Top of the chain must either handle the request or return the request.

Chain of Responsibility Pattern



@2014. Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Chain of Responsibility Pattern



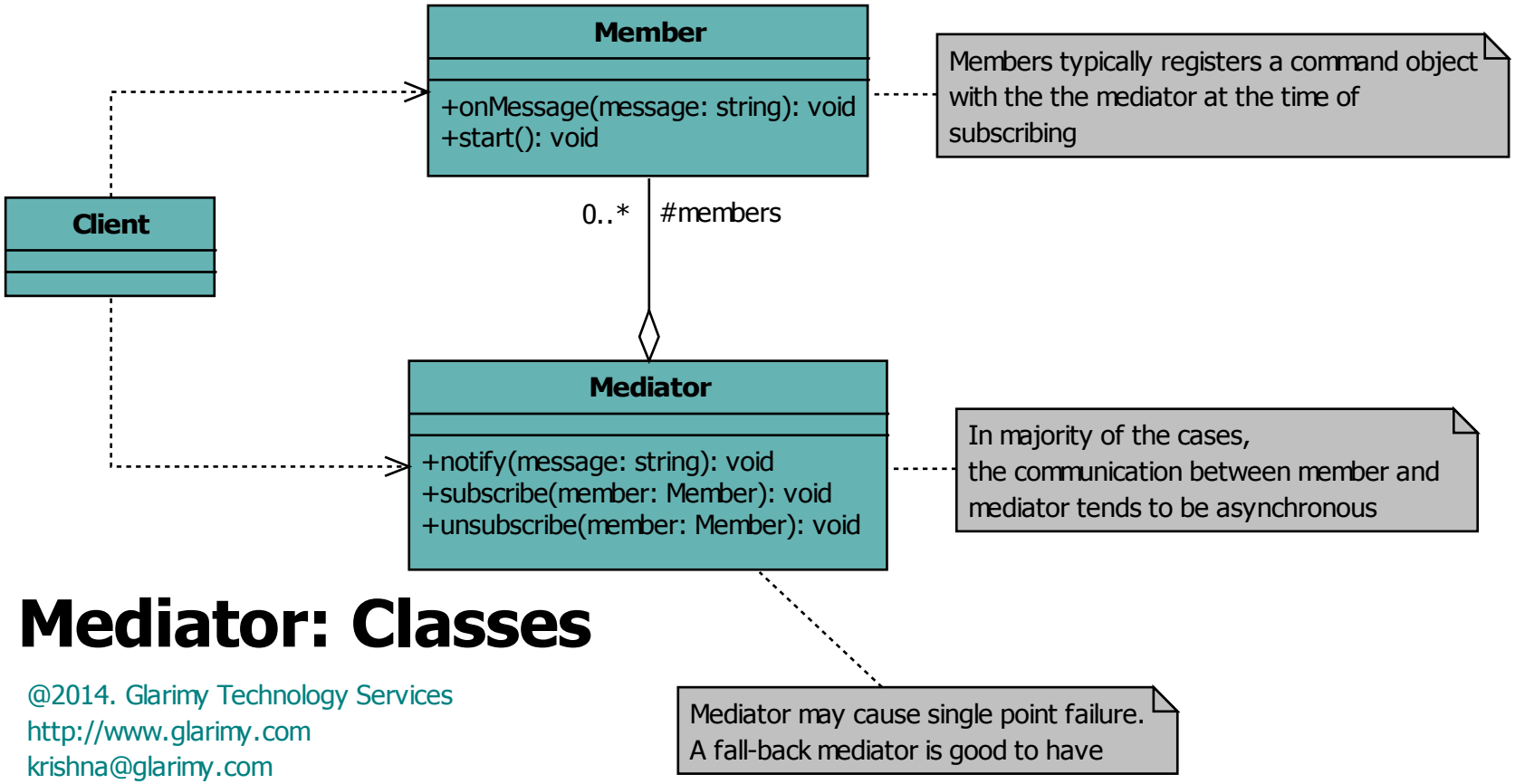
Chain of Responsibility: Interactions

@2014. Glarimy Technology Services | http://www.glarimy.com | krishna@glarimy.com

Mediator Pattern

- **Intent**
  - To enable broadcasting of information within a group
- **Description**
  - Scenario: Several objects may form a group. Any one of them may want to broadcast information to all other within the group. Any additions or omissions of the group members must not impact the rest in the group.
  - Scenario: Several objects may what to communicate but in a controlled and synchronized manner for consistency. Think of Air Traffic Control.
- **Implementation**
  - A special member called mediator within the group holds the reference to other member in the group. Every member pass the message to the mediator which in turn will be broadcasted to the group.
  - A fall back mediator also can be designated to avoid single-point failure.
- **Relations**
  - A group of mediators may form a chain of responsibility. When the information is flowing from outside of the group into the group, the mediator becomes an observer.

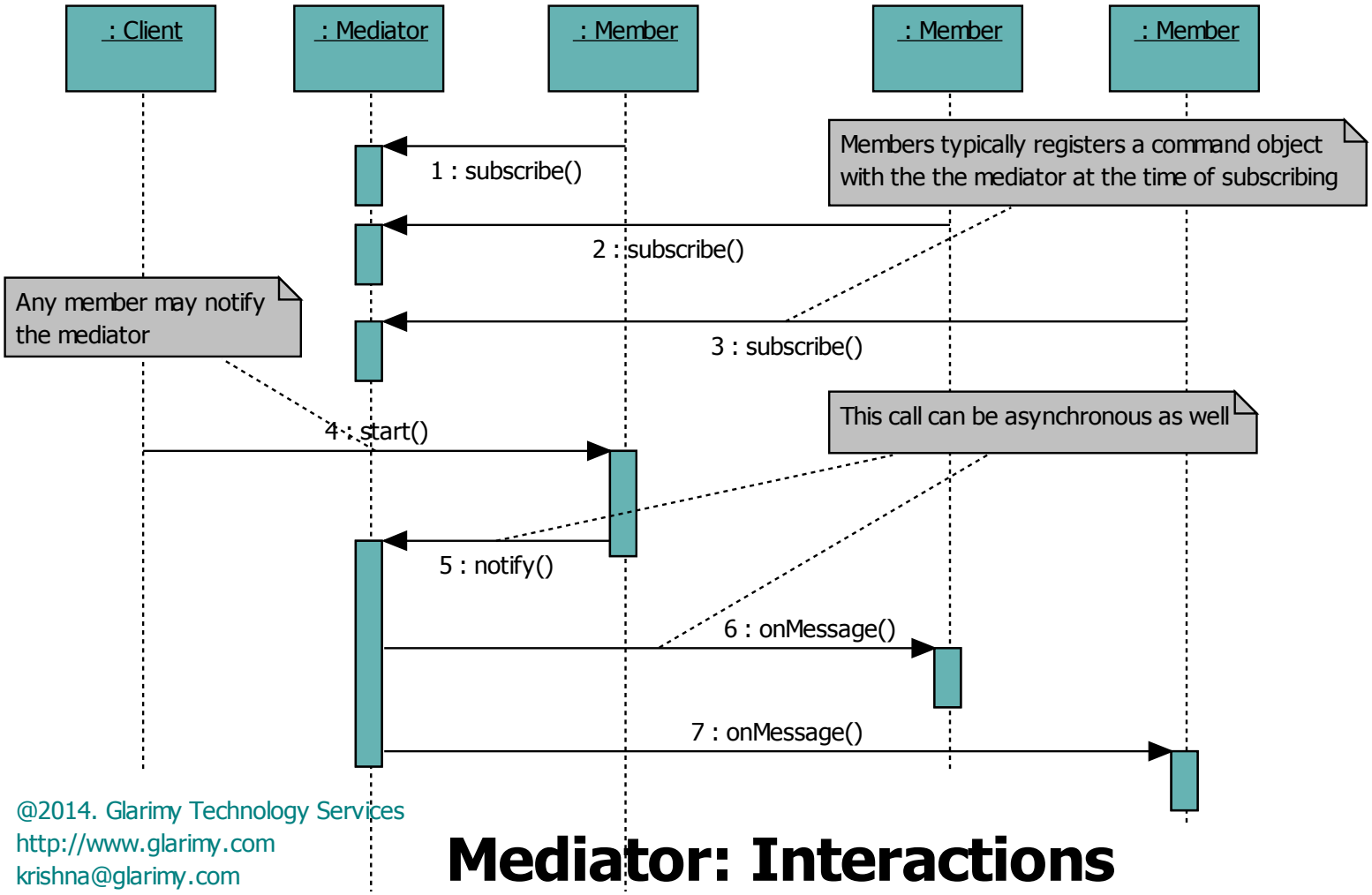
Mediator Pattern



Mediator: Classes

@2014. Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Mediator Pattern



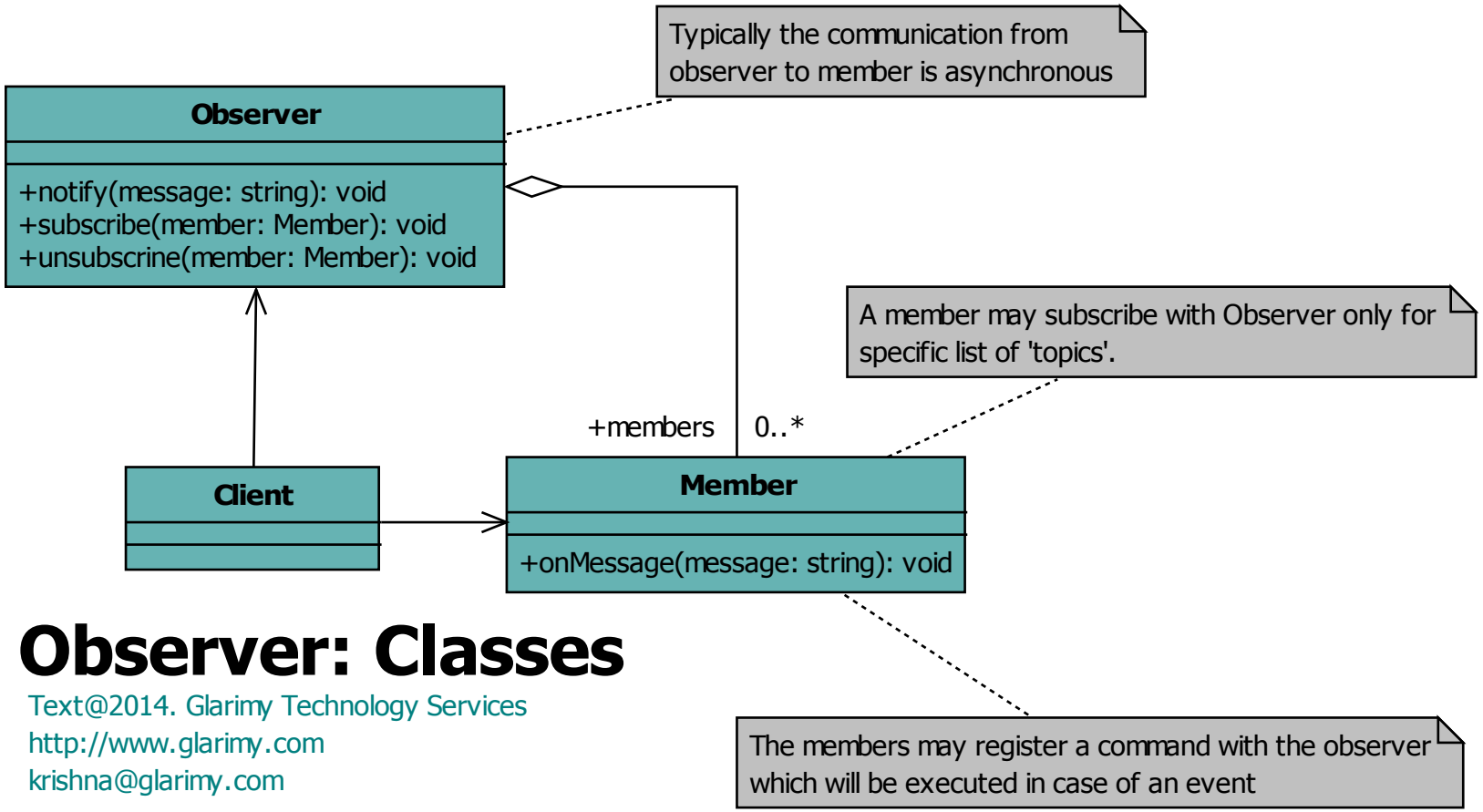
@2014. Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Mediator: Interactions

Observer Pattern

- **Intent**
  - To separate the responsibility of observing an outside event from the clients of the event.
- **Description**
  - Scenario: A group of objects are supposed to react to an external signal. However, the objects can not afford poll for the signal. Instead, an observer takes of tracking the signal on behalf of these objects. The observer alerts them when the signal takes place.
- **Implementation**
  - The clients subscribe for an event with the observer. Observer notifies all the subscribers in the event of signal.
- **Relations**
  - Observer may be given a command by the subscribers.

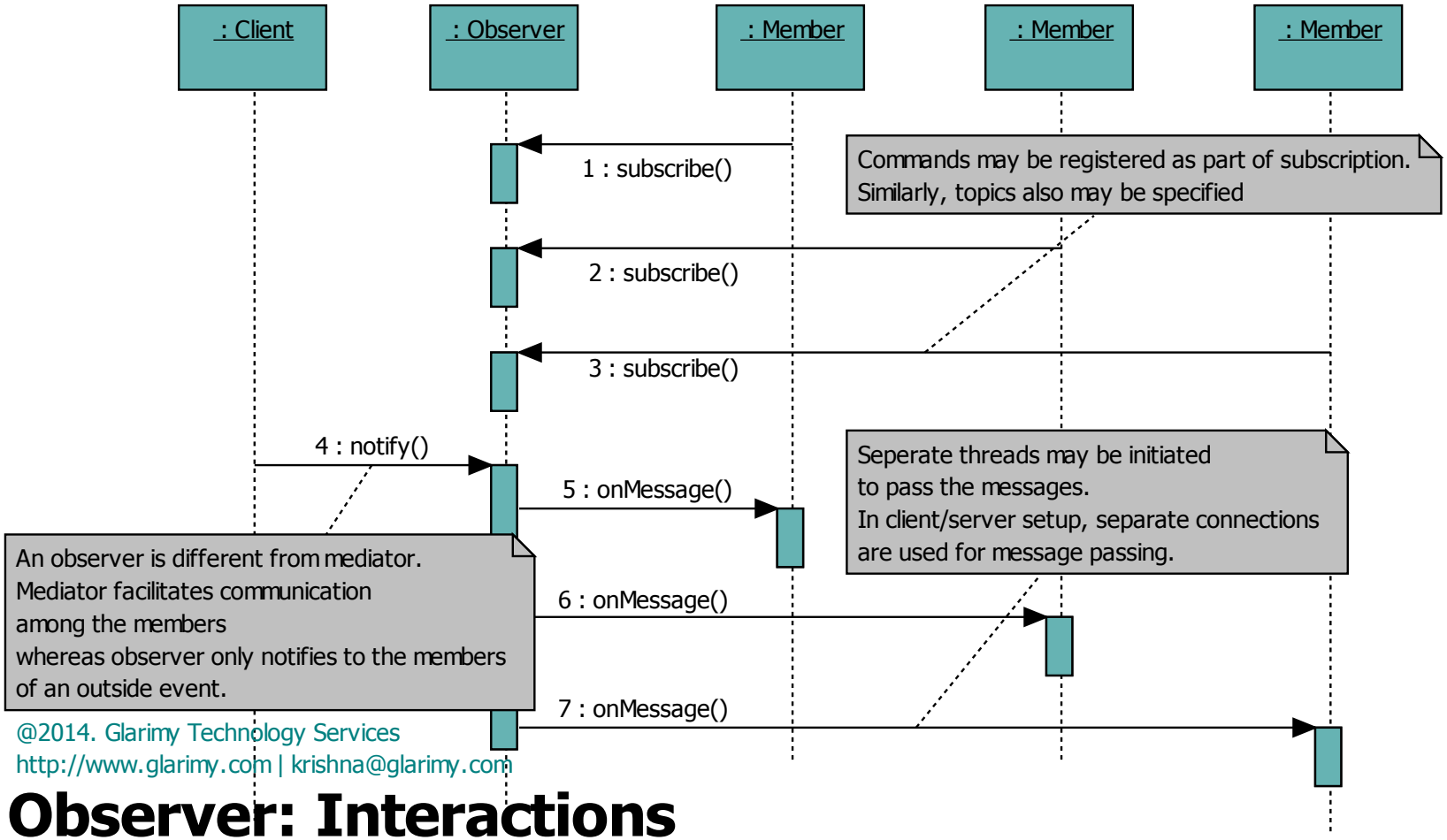
Observer Pattern



Observer: Classes

Text@2014. Glarimy Technology Services
http://www.glarimy.com
krishna@glarimy.com

Observer Pattern

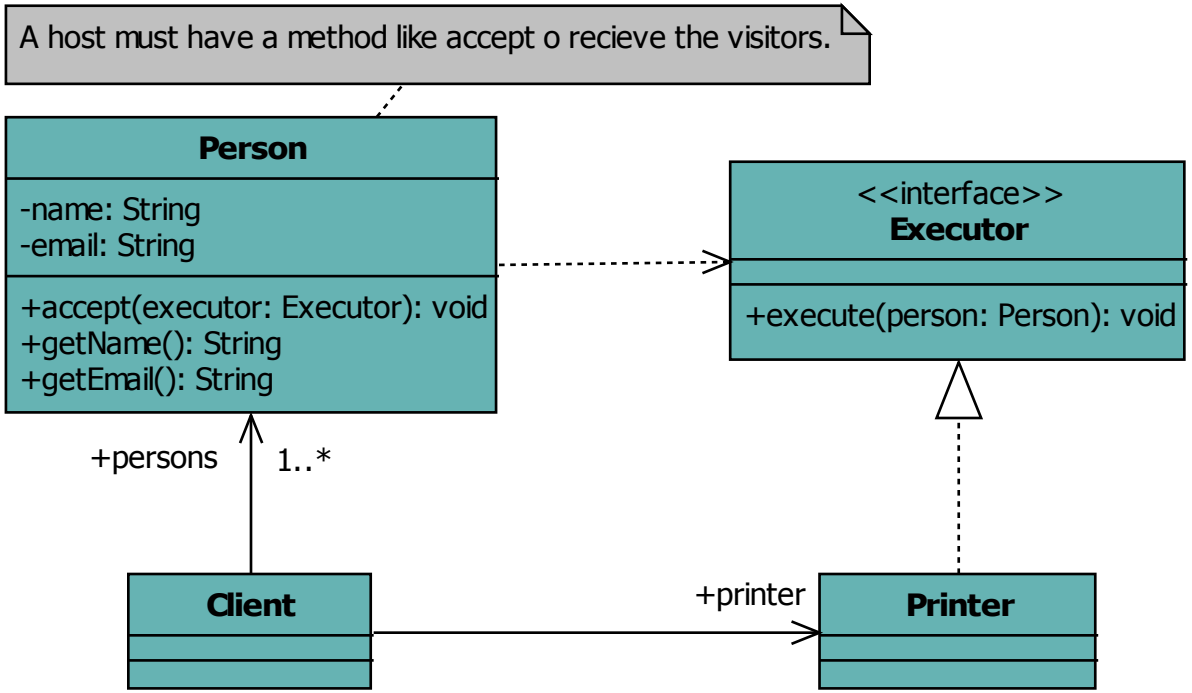


Observer: Interactions

Visitor Pattern

- **Intent**
  - To decorate several objects at runtime.
- **Description**
  - Scenario: An interface is implemented by several implementations or there are several objects instantiated from a give class. There is need comes to perform a common operation on all these objects, but such a behavior is not defined. A visitor visits all these objects to perform the operation as long as they accept the visitor.
- **Implementation**
  - The host objects that wants to be decorated implements an interface. The interface is equipped with a method to accept a visitor.
  - Visitors implement a common interface, but implemented to perform different operations on the hosts.
  - The host must provide access to the visitors in order for performing the operations.

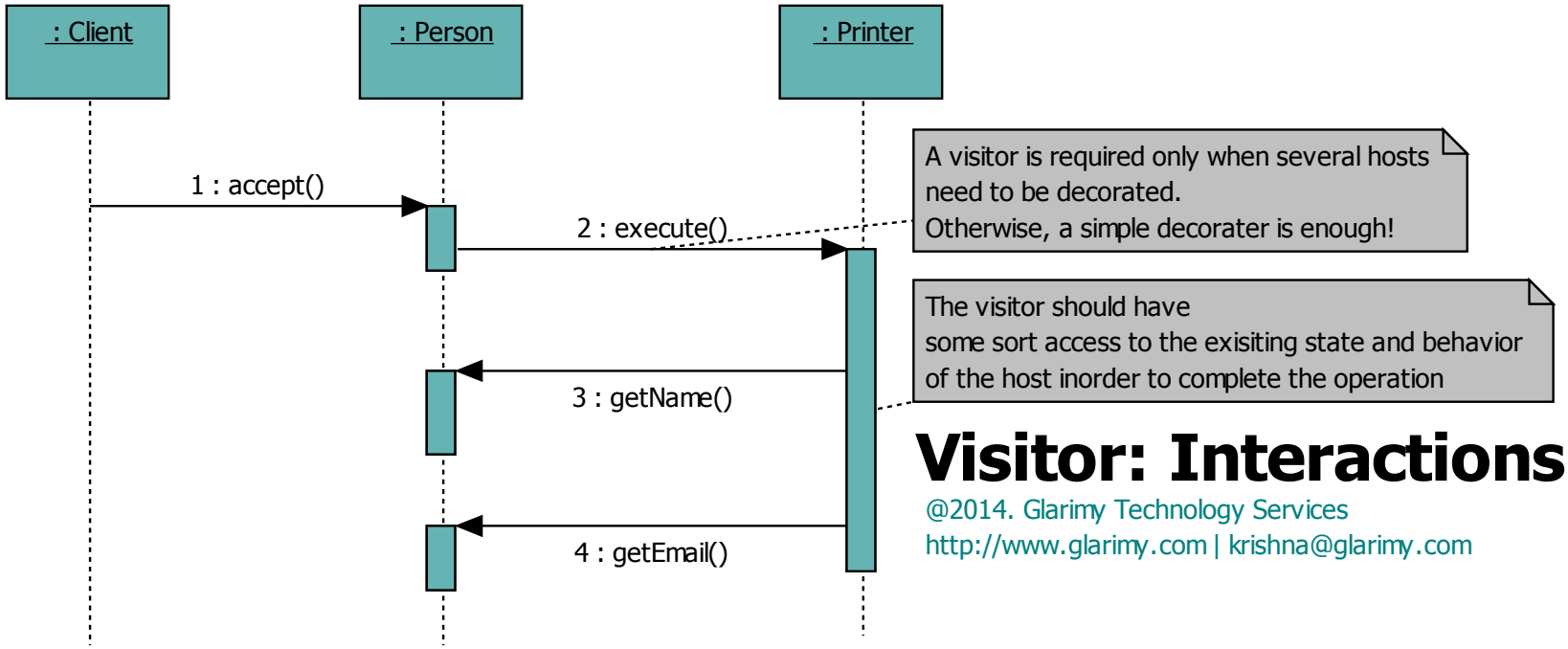
Visitor Pattern



Visitor: Classes

http://www.glarimy.com | krishna@glarimy.com

Visitor Pattern



Visitor: Interactions

@2014. Glarimy Technology Services
http://www.glarimy.com | krishna@glarimy.com

Glarimy Technology Services | Glarimy Technology Services | krishna@glarimy.com - Ya... | workshops.glarimy.com

GLARIMY

Technology Consulting
Corporate Training
Learning & Assessment Services

Glarimy Weekend Workshops @ INR 3000-00

JSP/Servlets/JSTL, Struts, JSF, GWT, Vaadin

Weekend Workshop Schedule | Booking Process

Weekend Workshops at Bangalore

Subscribe for **Schedule Updates** by sending a mail to workshops@glarimy.com. You will keep receiving updates on new learning material by mail.

Date	Title	Facilitator	Price	Availability
04-OCT-2014	Choose Yourself (?)	Krishna Mohan Koyya	INR 3,000-00	15 seats left
18-OCT-2014	Object Oriented Java Script Design Patterns	Krishna Mohan Koyya	INR 3,000-00	15 seats left
08-NOV-2014	Choose Yourself (?)	Krishna Mohan Koyya	INR 3,000-00	15 seats left
22-NOV-2014	C++ Design Patterns	Krishna Mohan Koyya	INR 3,000-00	15 seats left
06-DEC-2014	Choose Yourself (?)	Krishna Mohan Koyya	INR 3,000-00	15 seats left
20-DEC-2014	AngularJS	Krishna Mohan Koyya	INR 3,000-00	15 seats left

How to book at seat

Pay the requisite fee to GLARIMY TECHNOLOGY SERVICES, ICICI Bank Current A/C: 060105000973 (Old Madras Road Branch, Bangalore) by NEFT (IFSC Code: ICIC0000601)

Mail us the transaction number along with your name, contact number, employer name and your primary job role and workshop code to workshops@glarimy.com

You would receive confirmation mail with the receipt.

Free Resources

Principal Consultant

Krishna Mohan Koyya

View Full Profile

11 years of Development Experience at Cisco, Wipro and etc.,

5 years of training Experience for clients like HSBC, Bosch, Alcatel, Oracle and etc.,

1000 days of corporate trainings delivered since 2008

B.E in Electronics and Communications

M.Tech in Computer Science and Technology

Expertise in SOA, Patterns, UML, Web 2.0 (Dojo, ExtJS, JQuery) and JEE (Spring, Hibernate, EJB, JSF, OSGi)

http://workshops.glarimy.com | workshops@glarimy.com | Benguluru, Bharat (Bangalore, India)

Thank You

Glarimy Technology Services

Consulting & Corporate Training: http://www.glarimy.com

Weekend Executive Workshops: http://workshops.glarimy.com

Queries: krishna@glarimy.com

Twitter: @Glarimy